KARELIA-AMMATTIKORKEAKOULU
Tietojenkäsittelyn koulutusohjelma

Jani Kärkkäinen

PROSEDURAALISESTI GENEROIDUT METROIDVANIA-KENTÄT

Opinnäytetyö
Kesäkuu 2016

Karelia AMMATTIKORKEAKOULU

OPINNÄYTETYÖ
Kesäkuu 2016
Tietojenkäsittelyn koulutusohjelma

Karjalankatu 3
80200 JOENSUU
+358 50 468 3679

Tekijä(t)
Jani Kärkkäinen

Nimeke
Proseduraalisesti generoidut metroidvania-kentät

Toimeksiantaja
-

Tiivistelmä

2D-tasohyppelypelien ja varsinkin metroidvania-lajityypin kenttien proseduraalinen generointi rajoittuu usein templatingin käyttöön. Työssä tutkittiin käsinvalikoituja sisällöntuotanto- ja luolastongenerointi-algoritmeja. Työssä toteutettiin kyseiset algoritmit sekä kenttien toimivuuden testaukseen kehitettiin testiympäristö. Toteutus tehtiin Unity3D-pelimoottorilla Linux-käyttöjärjestelmässä.

Opinnäytetyössä verrattiin toisiinsa klassisen luolastogeneroinnin ja tutkimustyön tuloksena yhdisteltyjen algoritmien tuottamien kenttien tuloksia. Luotuja kenttiä verrattiin myös Metroidvania-kentistä analysoituihin kriteereihin.

Testiympäristö kehitettiin asteelle, jossa voidaan testata luotuja kenttiä pelinsisäisesti. Pelattavia kenttiä saatiin toteutettua molemmilla tavoilla. Yhdistellyt ja muokatut algoritmit olivat kuitenkin selkeästi rakenteellisempia ja pääsivät lähemmäksi Metroidvania-kenttämäisyyttä.

Muiden algoritmien vaikutus klassisen luolastogeneroinnin lisäksi jäi vajavaiseksi. Ainoastaan space colonization -algoritmilla oli merkittävä vaikutus. Algoritmeilla luotiin kenttägeometrialtaan toimivia kenttiä, mutta toimivan kenttäkokonaisuuden luomiseksi on jatkokehitys tarpeen.

Kieli	Sivuja 54
suomi	Liitteet 0

Asiasanat

proseduraalinen, generointi, algoritmi, unity3d, metroidvania

Karelia AMMATTIKORKEAKOULU

THESIS
June 2016
Business Information Technology

Karjalankatu 3
80200 JOENSUU
FINLAND
+358 50 468 3679

Author (s)
Jani Kärkkäinen

Title
Procedurally generated metroidvania levels

Commissioned by
-

Abstract

Procedural generation of levels in 2D-platformers and especially in metroidvania-games is usually restricted to templating. Hand-picked content creation and dungeon generation algorithms were analysed and implemented. To determine the viability of the created levels, a testing environment was implemented. Implementation was done on Unity3D game engine using Linux operating system.

The results of creating levels with both classical dungeon generation algorithms and combining and modifying analysed algorithms were compared with each other. Created levels were also compared against an analysis of Metroidvania levels.

The testing environment was developed so that the results could be tested. Playable levels were created with both of the methods. However, the combined and modified version generated more structural levels, and were also closer to Metroidvania-levels.

Using other algorithms for generating levels other than classical dungeon generation was not completely useful. Only the usage of space colonization algorithm provided any noticeable positive effect. To create a completely working level, more than just level geometry is needed, which requires more research and development. The results are however promising, and they are a good stepping stone for further work.

Language	Pages 54
Finnish	Appendices 0

Keywords

procedural, generation, algorithm, unity3d, metroidvania

Sisältö

1 Johdanto

Sisällöntuotantoalgoritmeja käytetään usein toteuttamaan monimutkaisia sisältöjä, kuten kohinakarttoja tai monimutkaisia orgaanisia järjestelmiä. Algoritmit on usein luotu tietyntyyppisen sisällön luontiin – esimerkiksi space colonization -algoritmi on luotu puiden laskennalliseen tuottamiseen. Algoritmeja voidaan soveltaa ja käyttää moniin muihinkin tarkoituksiin, esimerkiksi 2D-tasohyppelypelin kenttien luontiin, mikäli sopivat muutokset tai jatkokäsittelyt vain löydetään.

Nykyään onkin enenevässä määrin 2D-tasohyppelypelejä, jotka jollain tasolla luovat omat kenttänsä eri generointialgoritmeilla. Niissä on usein käytetty itse peliä varten suunniteltua algoritmia tai algoritmien yhdistelyä. Valitettavan usein ne kuitenkin pohjautuvat templatingiin eli mallikopiointiin, joka usein tuottaa toistavaa rakennetta.

2D-tasohyppelypelikenttien luominen proseduraalisesti on monivaiheinen prosessi. Kentän määritelmään usein kuuluu kenttägeometrian lisäksi myös vihollisten, tavaroiden sekä tasojen sijoittelu. Tässä työssä keskitytään pelkästään kenttägeometriaan.

2D-tasohyppelypeleistä olen valinnut alagenren, metroidvania. Sana on yhdistelmä pelien nimistä, Metroid ja Castlevania. Molemmille pelisarjoille on tyypillistä isot alueet, usein jopa vain yksi iso yhdistynyt kartta. Pelit keskittyvät huoneiden tutkimiseen, vihollisten kanssa taisteluun ja hahmonkehitykseen. Yksi iso kartta, jossa kaikki huoneet ovat saavutettavissa, tekevät Metroidvania-kenttien luomisen yleisillä sisällöntuotantoalgoritmeilla sopivammaksi.

Työssä tutkitaan muutamia algoritmeja, sekä yleisemmän sisällöntuotannon puolelta että ylhäältä kuvattujen pelien luolastogenerointimenetelmiin, ja

selvitetään, voiko näitä algoritmeja yhdistämällä ja muokkaamalla tuottaa kenttägeometrialtaan pelattavia Metroidvania-kenttiä.

Markkinoilla on tällä hetkellä hyvin vähän pelejä, jotka tuottavat Metroidvania-kenttiä, tai edes yleisesti 2D-tasohyppelipelikenttiä, proseduraalisesti ilman templatingia. Algoritmit ollaan pyritty valitsemaan niin, että ne tuottaisivat vähemmän toistavaa kenttärakennetta. Algoritmien soveltuvuus kenttien luomiseen arvioidaan useiden vuosien peliohjelmointityökokemuksen pohjalta.

Luotuja kenttiä varten toteutetaan testiympäristö. Testiympäristön avulla voidaan todentaa algoritmin tuottamat tulokset. Tämän työn keskeinen tarkoitus ei ole testiympäristön kehittäminen peliksi, joten sen graafinen ulkoasu, äänimaailma sekä muut pelilliset ominaisuudet, kuten edellä mainitsemani vihollisten ja tavaroiden sijoitus ja suunnitelu, voi jäädä hyvinkin vajavaisiksi. Tämä ei kuitenkaan haittaa, sillä testiympäristön tarkoituksena on vain todentaa tutkimus- ja kehitystyön tulokset.

1.1 Tavoite ja tutkimuskysymykset

Tavoitteena on yhdistellä esiteltyjä algoritmeja niin, että niistä saadaan luotua 2D-tasohyppelypelien, erityisesti metroidvania-tyyppisten pelien kenttänä toimivia kenttägeometrioita. Lisäksi analysoidaan, voisivatko pelkästään perinteiset ylhäältä kuvattujen luolastojen generointimenetelmät toimia metroidvania-kenttien luomisessa, vai antaisivatko muihin tarkoituksiin suunnatut, geneerisemmät generointialgoritmit mielekkäämpiä, paremmin analysoidut kriteerit täyttäviä rakenteita.

Lopuksi selvitetään miten valittuja algoritmeja täytyy muuttaa tai muokata jotta niillä voidaan tuottaa kenttiä, jotka täyttävät metroidvania-kentistä analysoidut kriteerit. Kriteereitä ovat muunmuassa kenttien läpäistävyys ja kiertorakenteet.

Nämä analysoidaan tarkemmin ennen toteutusta. Keskeisin tavoite on luoda algoritmeilla kenttägeometria, jota testataan runkopelillä.

Osaltaan työssä ja tutkimuksessa tavoitteena on selvittää henkilökohtaisen osaamisen taso; osaanko minä toteuttaa tällaisen työn? Toisaalta, juuri tällaista ei ole aikaisemmin tehty, tai ainakin hyvin harva on yrittänyt. Vastaavanlaisia yrityksiä ei ole tehty paljon, joten työssä pyritään myös selvittämään, onko 2D-tasohyppelypelien kenttägeometrian proseduraalinen generointi mahdollista tai saako siitä mitään lisäarvoa.

1.2 Työkalut

Kaikki työ, mukaan lukien raportointi, toteutetaan Linux-käyttöjärjestelmällä. Pääosa työkaluista on vapaata ohjelmistoa, kuten kirjoittamiseen käytetty LibreOffice tai ohjelmointiin käytetty VSCode. Algoritmien toteutus, lopullinen kenttägeometriaan luontiin tarkoitettu algoritmi sekä testiympäristö kuitenkin tehdään Unity3D-pelimoottorilla, joka on suljetun lähdekoodin kaupallinen ohjelmisto.

Unity3D-pelimoottori on kuitenkin ilmaiseksi käytettävissä ja sillä tehtyjä tuotoksia voi myydä tiettyyn tulorajaan asti. Pelimoottori on todella suosittu ja helppokäyttöinen, ja sillä voi todella nopeasti tehdä prototyyppejä. Tämän lisäksi olen itse käyttänyt kyseistä pelimoottoria päätoimessani jo useamman vuoden ajan. Näin työssä voidaan keskittyä algoritmien tutkimiseen ja toteuttamiseen uuden pelimoottorityökalun opettelemisen sijasta. Työn tärkeimmät tavoitteet ovat kuitenkin algoritmien tutkiminen, eikä se, millä työkaluilla tämä tutkiminen toteutetaan.

1.3 Odotetut tulokset

Odotukset ovat kolmijakoiset. Ensimmäinen odotettu tulos on saada yhdisteltyä esitellyt algoritmit ja menetelmät (pois lukien templating) niin, että niistä saadaan kenttägeometriaan liittyvää dataa ulos.

Toiseksi odotan yhdisteltyjen algoritmien avulla luodun kenttägeometrian olevan strukturoidumpi ja vähemmän sattumanvarainen kuin klassisella dungeon map -generoinnilla luotu kenttägeometria. Kolmas tulosodotus on testiympäristön toteutuksen onnistuminen niin, että sitä voi käyttää kenttägeometrian todennukseen, eikä tarvitse käyttää pelkästään debuggaamista varten visualisoitua dataa tulosten todentamiseen.

2 Dungeon map -generointimenetelmiä

Dungeon map -generointialgoritmeja käytetään pääosin roguelike-pelien kenttien eli luolastojen luomisessa. Roguelike-pelit ovat osaltaan hyvin samankaltaisia kuin metroidvania-pelit. Roguelike-peleissä pelin näkymä on kuitenkin ylhäältä alaspäin eikä sivusta kuvattu sekä kenttägeometria on usein väljempi.

Dungeon map -generoinnin voi pääosin jakaa kahteen erilliseen osioon: huoneiden luontiin ja tunneleiden luontiin. Templating on myös eräs dungeon map -generoinnin käytetyimmistä menetelmistä. Templating ei suoranaisesti liity kumpaankaan; sitä kuitenkin voidaan käyttää molemmissa. Templatingiä käytetään paljon, ja se johtaa usein selkeästi tietokonemaisiin ja itseään toistaviin geometrioihin.

2.1 Templating

Templating on tapa, jolla yhdistetään proseduraalinen generointi ja perinteinen sisällöntuotanto. Ensin suunnitellaan osia, joiden halutaan olevan mukana luodussa kentässä, ja sitten luodaan kenttä käyttäen eri algoritmeja niin, että kentälle asetettavat osat valitaan suunniteltujen osien joukosta.

Esimekiksi Diablo 3:ssa luodaan luolastot niin, että jokaiselle kentälle on muutama eri arkkityyppinsä. Näihin arkkityyppeihin on määritelty alueet, jotka luodaan proseduraalisesti, sekä alueet, joiden kohdille asetetaan jokin valmiiksi suunnitelluista palasista. Näin yhdistämällä saadaan sekä proseduraalisen generoinnin variaatiota että valmiiksi suunnitellun, tarkkaan mietityn ratkaisun hyödyt (kuva 1).

Kuva 1. Staattinen Diablo 3 -kenttä, jossa näkyvillä kohdat joihin laitetaan valmiiksi suunniteltuja osia satunnaisesti.

Myös kenttägeometria itsessään luodaan templatingillä: kenttää luodessa valitaan osakirjastosta osa, joka sijoitetaan karttaan, ja jatketaan luontia. Kartta luodaan siis kokonaan templatingillä: ensin kenttägeometria valmiiksi luoduista

osista, ja sitten valmiiksi suunniteltuja alueita ripotellaan kenttään, mikäli kenttään on luotu kohta johon niitä voi laittaa.

Muita templatingia käyttäviä pelejä on esimerkiksi Rogue Legacy ja Chasm. Nämä pelit käyttävät valmiiksi luotuja huoneita, joiden järjestystä vaihdellaan. Myös Abyss Odysseyssä käytetään templatingia yhteenliittämällä valmiiksi rakennettujen luolastojen osia.

Templatingin suurin ongelma on rakenteiden toistuvuus. Vaikka itse toteutetut osat olisivatkin mielenkiintoisia, ehkä jopa mielenkiintoisempia kuin proseduraalisesti generoidut on niiden toistuva käyttäminen puuduttavaa. Tätä voi lieventää tietysti tekemällä enemmän osia: mitä suurempi osien otanta, sitä vähemmän tulee toistuvuutta. Samalla toki työmäärä kasvaa.

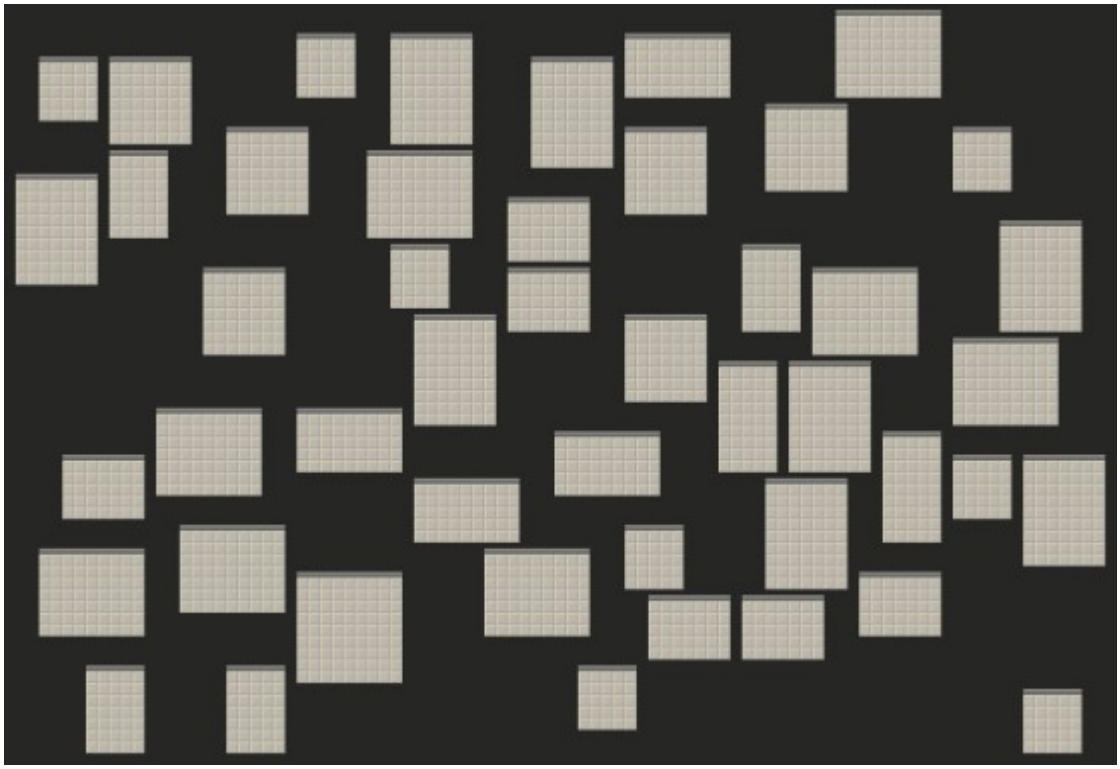
Parempi tapa olisikin kehittää algoritmia niin, että templatingiä ei tarvita. Metrdoivania-kenttiin sitä voisi kuitenkin käyttää mielenkiinnon lisäämiseksi satunnaisesti, esimerkiksi erikoistapahtumien, kuten erikoistavaran, tehtävän tai parannuspisteen, luontiin.

2.2 Huoneiden luonti

Huoneiden luonnissa keskitytään luolaston huoneiden määrään, kokoon ja sijaintiin. Yleisesti dungeon map -generoinnissa käytetään yksinkertaista algoritmia: luodaan satunnaisen kokoinen huone annetuilla määrittelevillä arvoilla satunnaiseen paikkaan, ja mikäli se menisi jo valmiiksi luotujen huoneiden päälle, yritetään uudestaan. Tätä toistetaan, kunnes luolastossa on tarpeeksi huoneita. Näin huoneiden asettelu on tehty esimerkiksi Angband-roguelike-pelissä. (Nystrom 2014.)

Tällaisessa huoneen luonnissa vaarana on mahdollisuus loputtomaan silmukkaan. Laskemalla huoneiden määrän ja koon raja-arvon oikein voidaan tätä välttää, mutta riski on kuitenkin olemassa. Parempi tapa on laskea

yrityskertoja. Esimerkiksi jos huoneenasettamisen yrityskertojen maksimimäärä on 200, yritetään luoda huonetta 200 kertaa luolastoon. Mikäli uusi huone onnistutaan luomaan, nollataan laskuri. Jos 200 kertaa täyttyy, luolaston huoneiden luonti on valmis (kuva 2). (Nystrom 2014.)

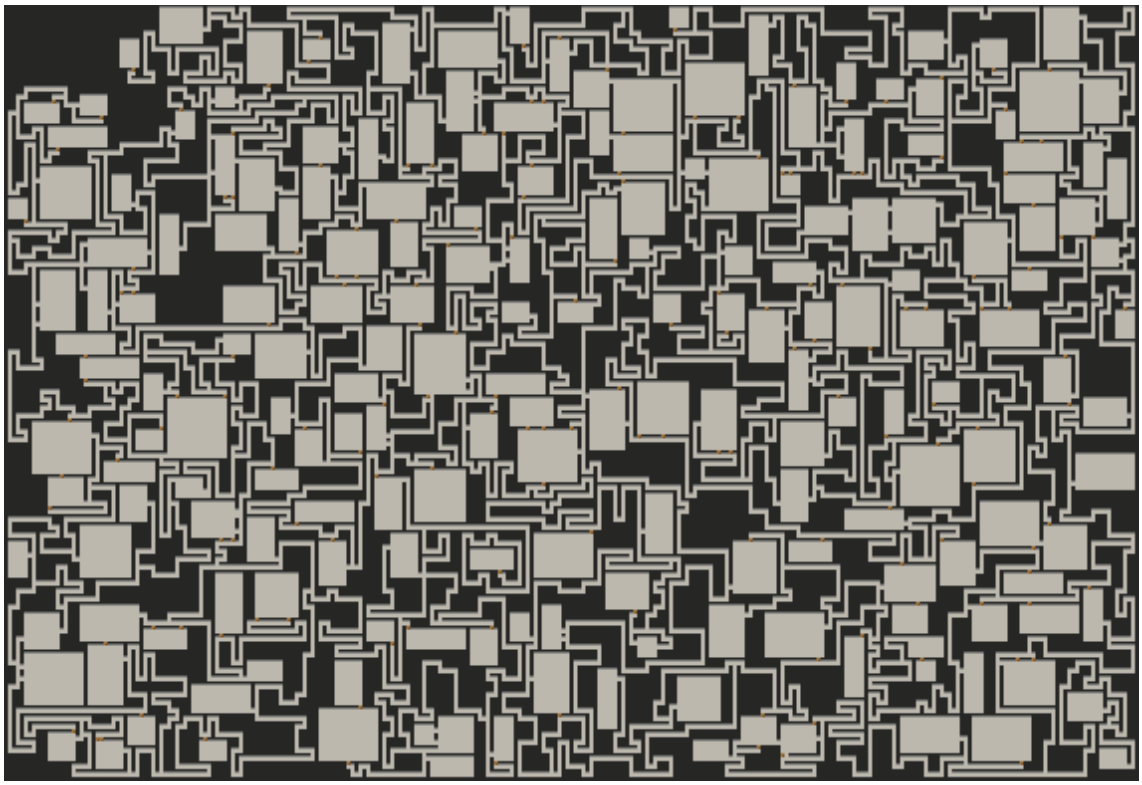


Kuva 2. Huoneiden asettelu dungeon map -generoinnissa (Nystrom 2014).

Huoneiden asettelua voisi käyttää metroidvania-kenttien luonnissa lähes sellaisenaan. Huoneiden sijainti on kuitenkin tällä tavoin luotuna todella strukturoimaton. Rakennetta voisi mahdollisesti kohdentaa paremmin metroidvania-kenttien luomiseksi käyttäen esimerkiksi space colonization -algoritmia, ja luoda huoneet näin saatujen järjestelmäpisteiden kohdille. Space colonization -algoritmiin voisi asettaa suuntarajoituksia, jotka lisäisivät metroidvania-kenttämäistä rakennetta.

2.3 Tunneleiden luonti

Pelkästään huoneita täynnä oleva luolasto ei ole vielä pelattavissa. Huoneet täytyy jollain tavalla yhdistää, jotta niiden välillä voi liikkua. Tunneleiden avulla pelaaja (ja viholliset) voivat siirtyä huoneista toisiin (kuva 3). Tunneleiden luomisessa käytetään usein jotain maze generation -algoritmia tai drunkard walk -algoritmia, joka on myös monen maze generation -algoritmin pohjalla (Buck 2011a).



Kuva 3. Luolaston huoneet yhdistetty tunneleilla. (Nystrom 2014.)

Maze generation -algoritmeja sokkelon luontiin on useita. Lähes kaikkien pohjalla kuitenkin toimii drunkard walk -algoritmi, jonka avulla määritellään usein sokkelon satunnaisuus. Käytännössä kaikkea satunnaista ruudukossa liikkumista, jossa kaikki mahdolliset suunnat ovat yhtä todennäköisiä, voidaan sanoa drunkard walk -algoritmiksi.

Drunkard walk -algoritmissa valitaan satunnainen sijainti, ja joka kierroksella liikutaan satunnaiseen suuntaan: ylös, alas, oikealle tai vasemmalle. Mikäli suunta johtaisi alueelta ulos, liikettä ei tehdä. Maze generation -algoritmeissa tätä käytetään niin, että mikäli solu johon mennään on käymätön, tehdään kulkuyhteys aikaisemman ja uuden solun välille. Parhaiten tämä tulee esille Aldous-Broderin algoritmissa, joka luo samalla todennäköisyydellä eri vaihtoehtojen joukosta sokkelon. (Buck 2011a.)

Recursive backtracking -algoritmissa drunkard walk -algoritmilla liikutaan satunnaisesti, ja merkataan reitti aina kun tulee uusi, käymätön sijainti, kuten Aldous-Broderin algoritmissakin. Suuntaa ei kuitenkaan valita kaikista suunnista, vaan ainoastaan läpikäymättömistä suunnista. Kun käy niin, että sijainnilla ei ole läpikäymättömiä naapureita, palataan taaksepäin. Sijainnin, jolla on vielä läpikäymätön naapuri, löydyttyä jatketaan algoritmia.

Recursive backtracking -algoritmi on suhteellisen nopea ja tuottaa mukavia algoritmeja. Algoritmi ei kuitekaan tuota kaikkia mahdollisia sokkeloita samalla todennäköisyydellä, ja sen tuottamissa sokkeloissa on usein pitkiä käytäviä. Recursive backtracking -algoritmi ei näin ollein sovellu kaikkiin tilanteisiin. (Buck 2011b.)

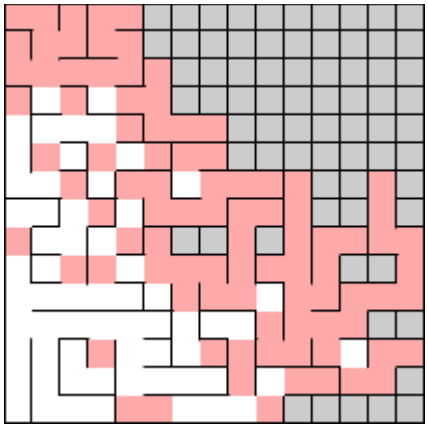
Primin algoritmilla, kuten muillakin maze generation -algoritmeilla, luodaan pienin virittävä puu, jota kaikki täydelliset sokkelot ovat. Hiukan muokattuna Primin algoritmin pystyy ottamaan käyttöön sokkeloiden luonnissa. Primin algoritmi usein tunnetaankin samannimisenä maze generation -algoritmina. (Buck 2011c.)

Primin maze generationiin muokattu algoritmi toimii niin, että jokaisella kierroksella valitaan reunasolujen - solut jotka ovat jo sokkelossa olevien solujen vierellä - joukosta satunnaisesti yksi solu. Tämä solu yhdistetään vieressä, jo sokkelossa olevaan soluun. Mikäli vieressä, jo sokkelossa olevia soluja on useita, valitaan niistä satunnaisesti yksi. Nyt kaikki viereiset, ei

sokkelossa olevat solut lisätään reunasolujen joukkoon, ja poistetaan solu, joka juuri lisättiin sokkeloon, reunasolujen joukosta. (Buck 2011c.)

Primin algoritmi on nopea ja muistitehokas. Primin algoritmi tuottaa kuitenkin helposti todella polveilevaa ja umpikujaista sokkeloa, joten se ei sovellu geneeriseksi algoritmiksi ihan kaikkialle. (Buck 2011c.)

Tällä hetkellä monipuolisin maze generation -algoritmi on growing tree -algoritmi (kuva 4), jolla voi aktiivisen solun – solu jolla on vielä prosessoimattomia naapureita - valintakriteeriä muuttamalla toteuttaa sekä recursive backtracking -algoritmin ottamalla uusimman solun että Primin algoritmin ottamalla satunnaisen solun. (Buck 2011a.)



Kuva 4. Growing tree -algoritmi keskivaiheilla. Punaiset solut ovat aktiivisten solujen listalla. (Buck 2011d.)

Mikäli metroidvania-kentät luodaan niin, että huoneet luodaan ensin, täytyy ne yhdistää jollain tavalla, jotta pelaaja pääsee niihin käsiksi. Maze generation -algoritmeja, ja varsinkin growing tree -algoritmia, voisi käyttää siihen vaiheeseen. Jos space colonization -algoritmilla luodaan huoneiden paikat, voitaisiin huoneita yhdistävät tunnelit luoda jokaiselle huoneparille erikseen growing tree -algoritmeilla. Huomioitavaa kuitenkin on, että mikäli space colonization -algoritmin kasvusuunnat rajoitetaan ruutupohjaiseksi, on se

itsessään ns. sparse maze generation -algoritmi: tällöin luodaan sokkelo, jonka kaikki ruudut eivät ole käytössä.

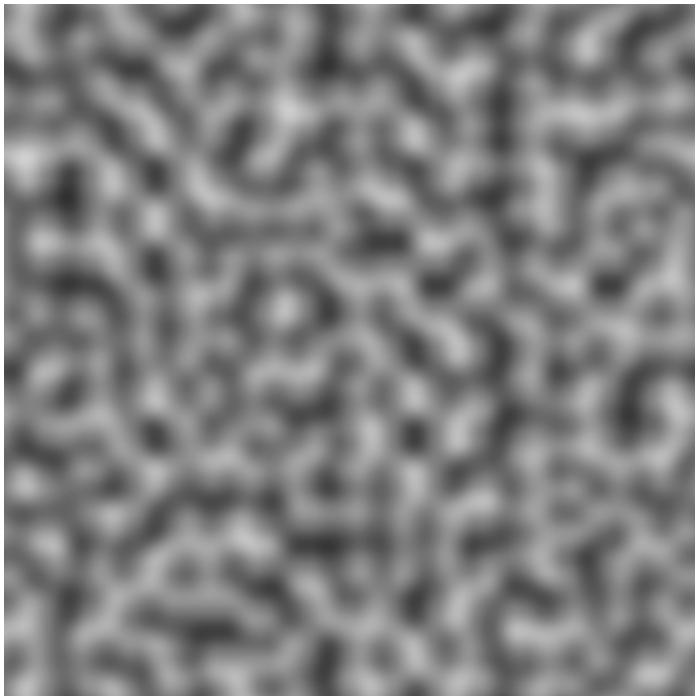
3 Yleisiä generointimenetelmiä

Yleisillä generointimenetelmillä tarkoitetaan tässä työssä kaikkia algoritmeja, joilla luodaan sisältöä, joka ei suoranaisesti liity peleihin tai tarkemmin metroidvania-tyyppisiin 2D-tasohyppelypeleihin. Yleisistä algoritmeista on valittu kolme algoritmia, jotka on ammattikokemuksen perusteella todettu sopivan metroidvania-kenttien luomiseen, mikäli niitä sovellettaisiin ja muokattaisiin.

Jokaisesta algoritmista esitellään itse algoritmi, siitä tulevat lopputulokset (esimerkikiksi kohinakartat, puurakenteet tai vaikka korkeuskartat) sekä sen, mihin algoritmia yleensä käytetään ja kuinka algoritmi toimii. Tämän jälkeen pohditaan, kuinka kyseisellä algoritmilla voisi toteuttaa metroidvania-kenttiä sekä kuinka paljon muutoksia algoritmiin joutuisi tekemään. Kolme algoritmia esitellään: Perlin noise -algoritmi, midpoint displacement -algoritmi sekä space colonization -algoritmi.

3.1 Perlin noise

Perlin noise -algoritmi on Ken Perlinin vuonna 1983 kehittämä gradient noise ("väriliukuinen kohina") -algoritmi. Kohina-algoritmia käytetään yleensä tietokonegrafiikan laskennalliseen luomiseen. Perlin noise -algoritmi voidaan laskea missä tahansa kokonaisluku-ulottuvuudessa, joskin se yleisimmin lasketaan kahdessa ulottuvuudessa (kuva 5). (Perlin 1999.)

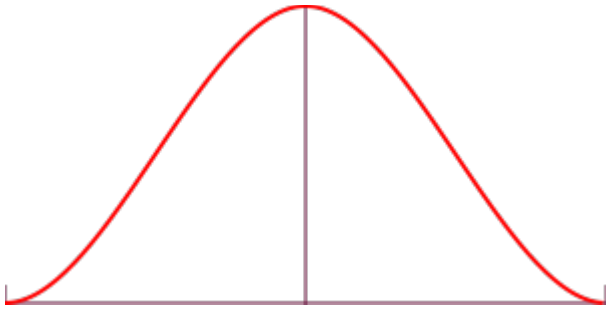


Kuva 5. Kaksiulotteinen Perlin noise -algoritmilla luotu kohinakartta (Kensler 2015).

Perlin työskenteli 1980-luvun alussa Tron-elokuvan parissa. Hän kyllästyi tietokoneella luotujen tekstuurien "tietokonemaisuuteen", ja kehitti oman kohina-algoritminsa, joka näyttäisi luonnollisemmalta. Tätä algoritmia käytettiinkin elokuvan proseduraalisten tekstuurien luomiseen. Perlin sai Academy Award -palkinnon vuonna 1997 työstään Perlin noise -algoritmin ja kohina-algoritmien kehittämisessä. 1980-luvun jälkeen Perlin noise -algoritmia ja muita kohina-algoritmeja on käytetty elokuvissa ja muussa mediassa enenevissä määrin, ja nykyään on hankala löytää valtavirran elokuvaa, jossa ei käytettäisi jotain kohina-algoritmia. (Perlin 1999.)

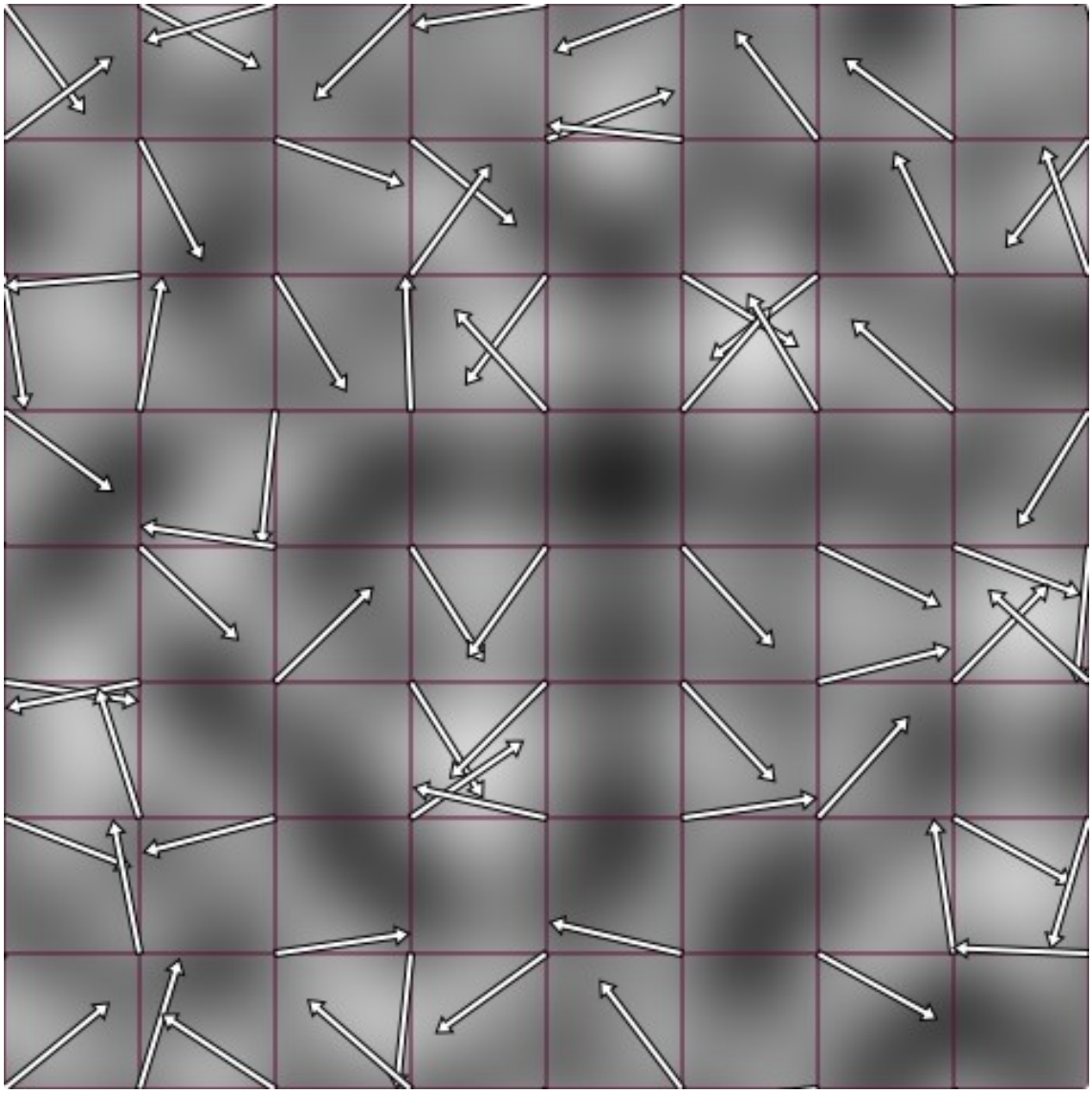
Perlin noise -algoritmia voidaan käyttää moneen eri tarkoitukseen, jossa tavoitteena on luonnollisen oloinen tuotos. Esimerkiksi savu- ja sumu-efektit, maasto, erilaiset pinnat (maalattu seinä, jne.) voidaan teksturoida kohina-algoritmia avuksi käyttäen. Näin niihin saadaan luonnollisen näköinen lopputulos.

Perlin noise -algoritmia luodaan ruudukkoon, jonka jokaiseen risteyskohtaan luodaan satunnainen väriliukuyksikkövektori. Jokaisen 2 x 2 -ruudukkoalueen arvot lasketaan käyttämällä interpolaatiofunktiota. Perlin noise -algoritmissa käytetään interpolaatiofunktiona peilattua Smoothstep-funktiota, joka on ajettu ajan itseisarvolla. Aika (t) on arvojen -1 ja 1 välillä (molemmat inklusiivisiä), jotta saadaan pehmeä ylösnouseva ja alaslaskeva käyrä (kuva 6). (Kensler 2015.)



Kuva 6. Perlin Noisen muokattu Smoothstep -interpolaatiofunktion kuvaaja (Kensler 2015).

Tämä interpolaatiofunktio ajetaan 2 x 2 -ruudukkoaluelle erikseen sekä x- että y-akselilla, ja kerrotaan tuloksena syntyvät arvojoukot keskenään. Tämän jälkeen kyseisen 2 x 2 -ruudukkoalueen risteyksessä olevan väriliukuvektorin sekä x- ja y-arvojen kesken lasketaan pistetulo, joka kerrotaan vielä interpolaatiofunktion kanssa (kuva 7). (Kensler 2015.)



Kuva 7. 2D Perlin noise -algoritmin luoma kohina, havainnollistettu ruudukon ja väriliukuvektorien kanssa (Kensler 2015) .

2D-tasohyppelypelien tasonluontiin algoritmilla ei ole suoraa verrattavuutta. Perlin noise -algoritmia voisi käyttää ainakin siihen, että loisi ns. "kuumia pisteitä" tueksi joko jollekin toiselle algoritmille, tai suoraan kentässä käytettävän liiketason luomiseksi.

Ruudukon kokoa säätämällä pystyisi määrittelemään alueiden koon. Yhdistelemällä kahta tai useampaa generoitua kohinaa keskenään, voisi tehdä erilaisia yksityiskohtia. "Kuumiin pisteisiin" voi luoda erilaisten sääntöjen –

esimerkiksi että yksikään taso ei saa olla toista tasoa pelihahmon korkeutta korkeammalla – avulla tasoja, joilla pelaaja voi hyppiä ja liikkua.

Kohinaa ja niiden alueita voi myös käyttää pohjittamaan esimerkiksi myöhemmin esiteltävää space colonization -algoritmia. Space colonization -algoritmi luo ns. reittiä, joka polveilee ja haarautuu. Se tarvitsee toimiakseen pistepilven, jotta algoritmi "löytää" oikean kohdan minne seuraavaksi reittiä viedä. Pistepilven käyttö simuloi orgaanista kasvamista, jossa kasvetaan sinne, missä on otollisimmat kasvualueet.

Perlin noise -algoritmilla voisi määritellä, miten pistepilven kontrollipisteet muotoutuvat: Perlin noise -algoritmilla generoitu 2D-kartan piste voisi toimia suoraan todennäköisyytenä siihen, muodostuuko kyseiseen kohtaan piste vai ei. Tämä ohjaisi todennäköisesti space colonization -algoritmia muodostamaan mitä mielenkiintoisempia reittejä.

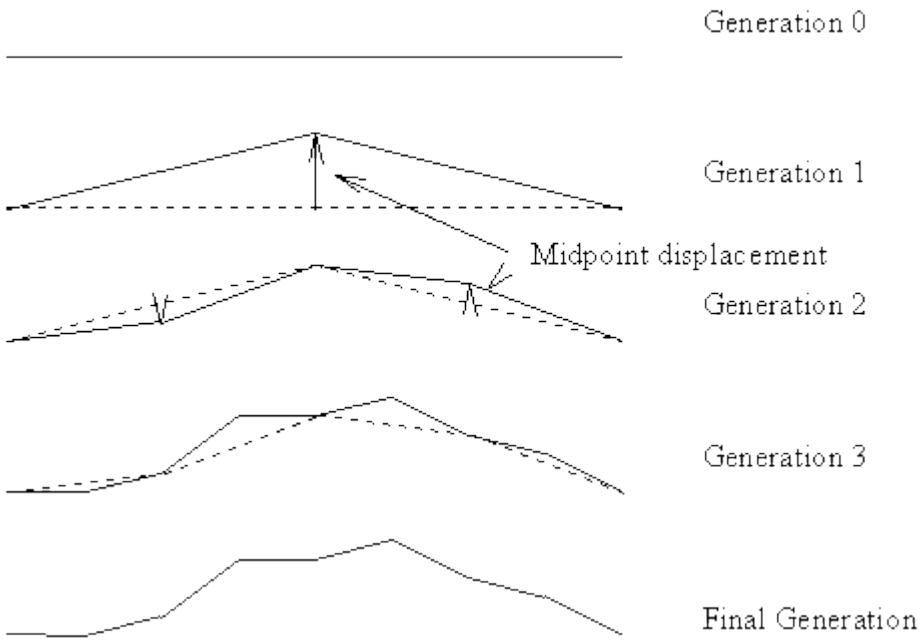
3.2 Midpoint displacement

Ensimmäisiä kertoja midpoint displacement -algoritmia käytettiin jo 1920-luvun alkupuolella fraktaalimatematiikan saralla. Myöhemmin algoritmia on käytetty useaan otteeseen tietokonegrafiikassa. (Tsai 2003.)

Midpoint displacement -algoritmia ei pidä sekoittaa Diamond-Square-algoritmiin (Fournier, Fussel & Carpenter 1982), jolla voidaan luoda 3D-maastoa. Usein näin kuitenkin käy, sillä Diamond-Square-algoritmi käyttää hyvin samankaltaista menetelmää. Diamond-Square-algoritmi tunnetaan myös random midpoint displacement fractal -algoritmina, ja se voidaankin mieltää kolmanteen ulottuvuuteen laajennetuksi midpoint displacement -algoritmiksi.

Algoritmin toimintatapa on yksinkertainen: kahden pisteen välillä on jana. Tämän janan keskelle luodaan piste, jonka korkeutta muutetaan satunnaisella arvolla, joka luodaan tietylle lukuvälille. Seuraavaksi lukuväliä pienennetään, ja

jaetaan nyt muodostuneet janat samalla tavalla kuin ensimmäinenkin. Tätä jatketaan, kunnes tarvittava määrä iteraatioita on suoritettu (kuva 8). (Johnson 2016.)



Kuva 8. Midpoint displacement -algoritmin vaiheet (Johnson 2016).

Midpoint displacement -algoritmilla voidaan luoda fraktaaliviivaa, joka voi esimerkiksi kuvantaa rantaviivaa. Midpoint displacement -algoritmilla luodaan myös esimerkiksi vuorien siluettia taustalle.

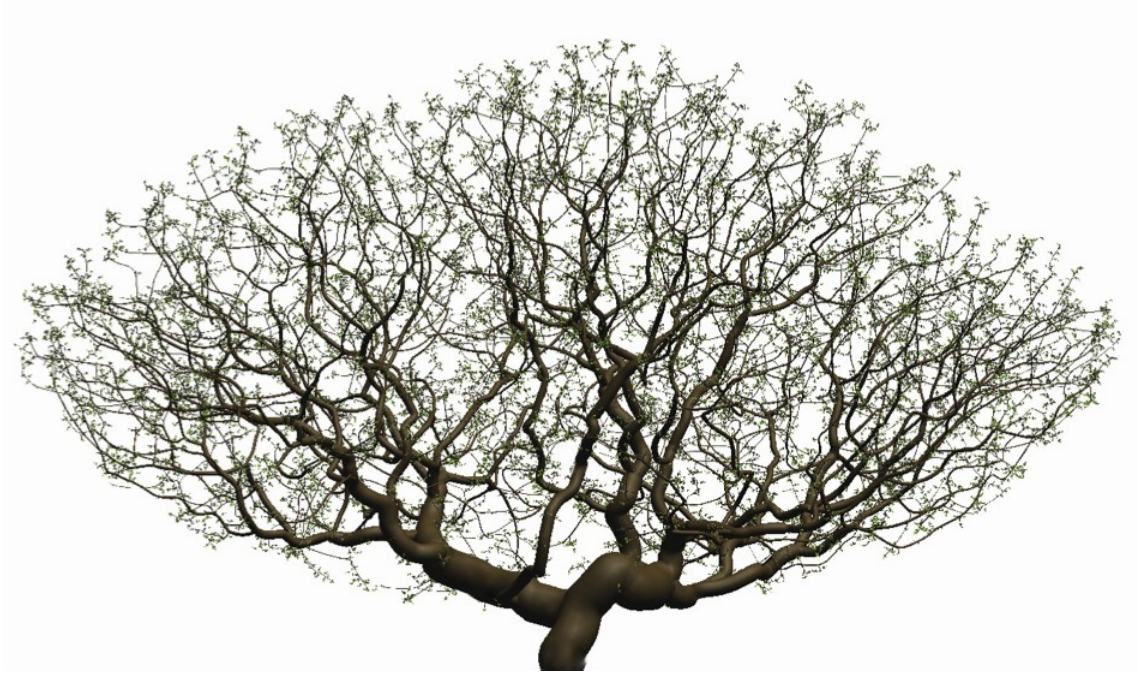
Algoritmin käyttö 2D-platformereissa saattaa helposti rajoittua juurikin taustagrafiikan generointiin. Esimerkiksi parallax-scrolling yhdistettynä kolmeen tai useampaan tällaiseen vuorisiluettiin niin, että eri tasot on värjätty eri tavalla, saisi aikaisiksi todennäköisesti silmää miellyttävän dynaamisen taustan.

Midpoint displacement -algoritmia voisi kuitenkin käyttää myös kenttägeometrian luomiseen: esimerkiksi luolasto-aiheisissa kentissä kenttägeometrian voisi hyvinkin uskottavasti luoda midpoint displacement -algoritmilla. Tässä tapauksessa algoritmia voisi muokata niin, että muutokset ja puolivälikohdat katsottaisiin janasuuntaisella kuvitteellisella akselilla, sen sijasta

että se katsottaisiin x- ja y-akselilla. Tämä todennäköisesti lisäisi elävyyttä sekä toisi luotuun kuvioon mukaan niin kutsuttuja taskuja, joka kuvastaisi luolatilaa paremmin.

3.3 Space colonization

Space colonization -algoritmia (Runions, Lane & Prusinkiewicz 2007) käytetään erilaisten orgaanisten järjestelmien, kuten verisuoniston, tieverkoston tai hermojärjestelmän, mallintamiseen. Yleisimmin sitä käytetään kuitenkin puiden (kuva 9) generointiin.



Kuva 9. Esimerkki space colonization -algoritmilla luodusta puusta (Runions ym. 2007).

Space colonization -algoritmi on jatkokehitetty open leaf venation pattern -algoritmista (Runions, Fuhrer, Lane, Federl, Rolland-Lagan & Prusinkiewicz 2005), joka toimii vain kahdessa ulottuvuudessa. Space colonization -algoritmi toimii sekä kolmessa että myös kahdessa ulottuvuudessa, mikäli algoritmin

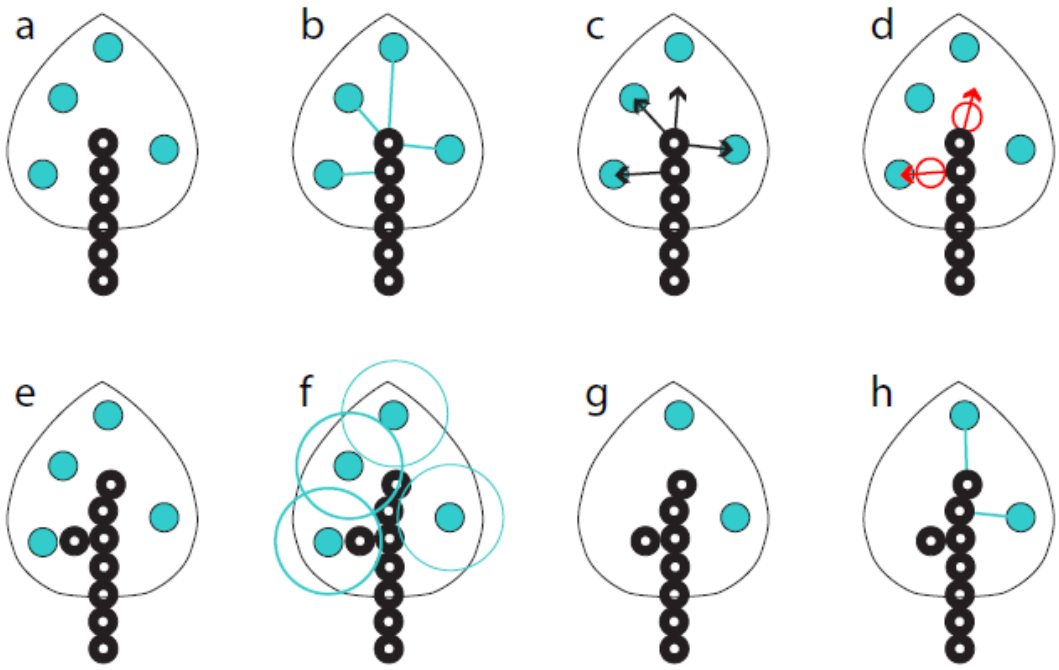
parametrinä toimivan peittoalueen kokoa säätää niin, että jokin ulottuvuuksista on 0.

Algoritmin toiminta simuloi tilannetta, jossa tietty kasvava, haarautuva yksikkö kilpailee itsensä (tai miksei muidenkin) kanssa tilasta. Esimerkiksi puut ja kasvit kilpailevat auringonvalosta, tieverkostot positiivisista maasto-olosuhteista ja niin edelleen. Space colonization -algoritmi voidaan jakaa kolmeen vaiheeseen: peittoalueen luontiin, houkuttelupisteiden vaikutuksen järjestelmäpisteisiin laskemiseen sekä houkuttelupisteiden poistoon.

Ensin luodaan peittoalue. Peittoalueen kokoa ja muotoa muuttamalla voidaan määritellä lopullisen tuloksen muotoa. Peittoalueen sisälle luodaan joukko houkutuspisteitä. Ensimmäisellä iteraatiolla verrataan puun (tai muun järjestelmän) ensimmäisen pisteen ja houkutuspisteiden etäisyyttä. Houkutuspiste vaikuttaa ainoastaan lähimpää järjestelmäpistettä, vaikka se olisikin muuten järjestelmäpisteen vaikutusetäisyydellä. Mikäli se on vaikutusetäisyydellä, se lasketaan mukaan joukkoon, jotka houkuttelevat puu- eli järjestelmäpistettä (kuva 10a,b).

Kaikista järjestelmäpistettä houkuttelevista pisteistä lasketaan normalisoitu vektori suhteessa järjestelmäpisteeseen, ja näistä kaikista vektoreista lasketaan keskiarvoinen suuntavektori. Suuntavektorin avulla luodaan uusi piste. Etäisyys uuden pisteen ja vanhan pisteen välillä on järjestelmän sisällä vakio (kuva 10b,c,d,e).

Lopuksi kaikki houkuttelupisteet, jotka ovat tappoetäisyyden päässä järjestelmäpisteistä, tuhotaan (kuva 10f,g). Näin saadaan merkattua, että kyseinen alue on nyt vallattu: tila ja resurssit on otettu käyttöön. Tällä tavoin algoritmi simuloi luonnossa esiintyviä järjestelmiä.



Kuva 10. a,b) Peittoalue, houkuttelupisteet (sininen) ja järjestelmäpisteet (musta) b, c) Houkuttelupisteet vaikuttavat järjestelmäpisteisiin d,e) Uudet järjestelmäpisteet luodaan g,h) Liian lähellä olevat houkuttelupisteet poistetaan ja toistetaan toiminnot. (Runions ym. 2007.)

Koska space colonization -algoritmi on monimutkaisempi kuin aikaisemmin esitellyt algorimit, sen käyttö 2D-kenttien luomisessa ei ole aivan niin suoraviivaista. Yksin ja muokkaamattomana käytetty space colonization -algoritmi on todennäköisesti vaikeasti käytettävissä tällaisessa tapauksessa. Mikäli algoritmia kuitenkin muokkaa tai yhdistää muihin algoritmeihin, voi lopputuloksesta tulla erittäin dynaamista ja orgaanista.

Jos järjestelmäpisteet jättää hyvin väljiksi, voisi järjestelmäpisteiden paikkatietoa käyttää hyväksi kentän luonnissa. Järjestelmäpisteiden kohtiin voisi luoda vaikkapa huoneita. Vaihtoehtoisesti järjestelmäpisteitä voisi käyttää kohtina, joihin luotaisiin taso, jolla pelaaja voi kävellä – muita algoritmeja käyttäen tehtäisiin muu kenttägeometria. Tällä tavoin käytettynä algoritmin lopputulos toimisi liikekarttana pelaajalle, ja se mahdollistaisi enemmän pelimekaniikkaan keskittyvän generoinnin kuin perinteinen kenttägeometrian luonti.

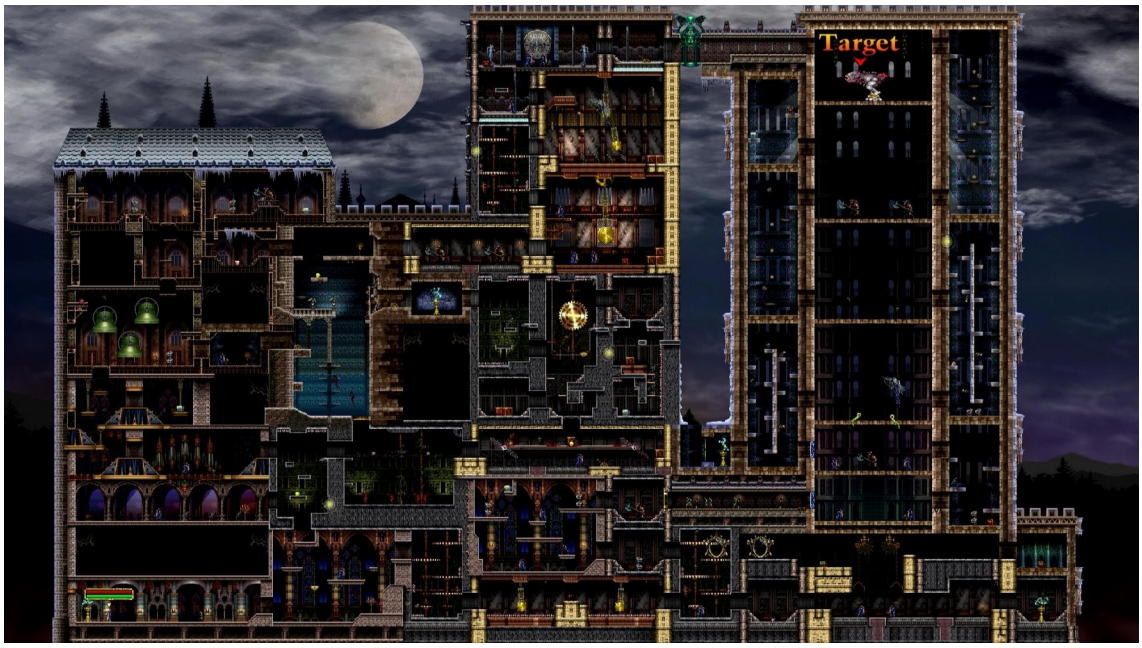
4 Toteutus

Kaikki toteuttaminen tapahtui Unity Game Engine -ympäristössä. Toteutus tehtiin Linux-ympäristössä käyttäen Unity Editoria sekä VSCode -koodieditoria.

Ensin toteutettiin kaikki työssä mukana olevat algoritmit. Algoritmit toteutettiin niin, että ne antoivat ulos mahdollisimman geneeristä dataa, jota voisi käyttää helposti eri tavoin, ja myös tarpeen vaatiessa yhdistellä keskenään. Testiympäristö kehitettiin algoritmien lopputulosten testaamiseksi. Metroidvania-kenttia analysoitiin ja lopputuloksia verrattiin analysoituihin kriteereihin.

4.1 Metroidvania-kenttien rakenne

Metroidvania-kentät ovat usein sokkelomaisia, monihuoneisia komplekseja. Paikoin huoneet ovat todella pitkulaisia, tosin useimmin ne tuntuvat noudattavan perusneliön muotoa (kuva 11).



Kuva 11. Tyypillinen Metroidvania-kenttä, pelistä Castlevania: Harmony of Despair.

Metroidvania-kentissä ei usein erotella tilaa huoneen ja huoneita yhdistävän käytävän välillä, vaan nämä käytävät ovat itsessään huoneita. Kartan luominen proseduraalisesti kuitenkin helpottuu huomattavasti, mikäli luo erikseen huoneet ja käytävät, ja haasteena onkin keksiä, kuinka luoda nämä huoneita yhdistävät rakenteet niin, etteivät ne vaikuta suoranaisesti "vain" käytäviltä.

4.2 Kriteerit

Metroidvania-kentät ovat sokkelomaisia, ja vaikka käytävätkin ovat usein lähes erottamattomia huoneista itsestään, voidaan kentän rakenne jakaa toteuttamistarkoituksessa huoneisiin ja käytäviin. Näin ollen lopullisen algoritmin pitäisi tuottaa huoneita, jotka on yhdistetty käytävillä.

Metroidvania-kentät, vaikka ovatkin sokkelomaisia, eivät kuitenkaan ole niin kutsuttuja täydellisiä sokkeloita, eli ne tekevät kiertorakenteita. Normaalisti esimerkiksi maze generation -algoritmit tuottavat vain täydellisiä sokkeloita, eivätkä ne tee kiertorakenteita. Lisäkriteerinä olisi siis kiertorakenteiden luonti.

Kaikkien kenttien täytyy myös olla niin sanotusti "läpäistävissä". Koska työssä keskityttiin ainoastaan kenttägeometriaan, testiympäristön täytyy pystyä luomaan visuaalinen kenttägeometria ja sen tarvittavat fysiikkamoottorin objektit Keskittymisen ollessa kenttägeometriassa, voi testiympäristö olla myös huomioimatta niin sanotut pelitekniset ominaisuusvaatimukset. Testiympäristö siis voi tarvittaessa toteuttaa matkustamisen eri huoneiden välillä vaikka niin, että ilmassa hyppiminen on mahdollista, ja näin voidaan keskittyä vain huone- ja käytäväarakenteen rakentamiseen.

4.3 Kehityskaari

Totetuksessa lähdettiin ensin testiympäriston pohjan toteuttamisesta. Näin algoritmeja toteutettaessa olisi jotakin jonka päälle rakentaa ne. Testiympäristön

pohjan jälkeen algoritmit rakennettiin yksitellen omiin scene-tiedostoihinsa. Toteutuksessa otettiin huomioon myös algoritmien käyttö muissa scene-tiedostoissa, ja näin ollen ne pyrittiin toteuttamaan mahdollisimman riippumattomiksi.

Algoritmien toteutuksen jälkeen keskityttiin implementoimaan testiympäristössä käytettävä kenttägeometria. Ensin toteutettiin klassinen luolastogenerointi. Klassisen luolastogeneroinnin toteutuksen valmistuttua yhdistettiin geneerisemmät sisällöntuotantoalgoritmit klassiseen luolastogenerointiin.

Seuraavaksi implementoitiin runkopeliin jo ensimmäisessä vaiheessa pohjalle rakennettujen mekanismien avulla kenttägeometrian pelinsisäinen luonti. Kenttien testaus oli tässä vaiheessa jo mahdollista testiympäristön avulla. Näin saatujen kenttien toimivuutta analysoitiin. Eri tavoin luotujen kenttien vastaavuuttaa verrattiin metroidvania-kentistä analysoituihin kriteereihin.

4.4 Algoritmit

Jokaista algoritmia kohden tehtiin ensin viitetoteutus. Tästä johtuen ne eivät ole optimoituja, mutta soveltuvat parhaiten niiden yhdistelmien tutkimiseen, koska ne on toteutettu algoritmien kuvausten mukaisesti.

Jokainen algoritmi on toteutettu kaksijakoisesti. Jokaiselle algoritmille on oma Unityn scene-tiedosto, jossa on GameObject jonka komponentiksi määritellään algoritmin Controller-skripti. Controller-skripti ohjaa ei-Unity-komponentoitavaa (vaikkakin Unity-sidonnaista) algoritmi-luokkaa. Toteutuksessa eroteltiin kuitenkin scene-tiedosto itse algoritmin toteutuksesta niin, että se ei ole riippuvainen scene-tiedostosta tai Unityn MonoBehaviour-komponenttiluokasta. Näin algoritmit saatiin käyttöön myös muualla projektin sisäisesti, eivätkä ne olleet sidottuna tiettyyn GameObject-instanssiin tai scene-tiedostoon.

Algoritmit itsessään toteutettiin omiin erillisiin luokkiinsa. Luokkien käyttöideana oli aina uutta lopputulosta laskiessa luoda uusi instanssi kyseisestä luokasta. Monet toteutetut luokat käyttävät Unityn tarjoamia luokkia paljon, kuten Vector2, joka on kaksiulotteinen vektoriluokka.

Maze generation -algoritmeista valitsin toteutettavaksi growing tree -algoritmin. Valinta perusteltiin algoritmin toteuttamisen yksinkertaisuudella sekä algoritmin säätömahdollisuuksien sekä lopputulosten monipuolisuudella.

Algoritmien yksittäisen toteutuksen jälkeen yhdisteltiin algoritmit toisiinsa. Pyrkimyksenä oli toteuttaa kriteerit paremmin toteuttavaa lopputulosta kuin pelkällä dungeon map -generoinnilla (eli yhdistelemällä dungeon room placement -generointi maze generation -algoritmin kanssa), joten tarpeen vaatiessa algoritmeja myös muokattiin paremmin tilanteeseen sopiviksi.

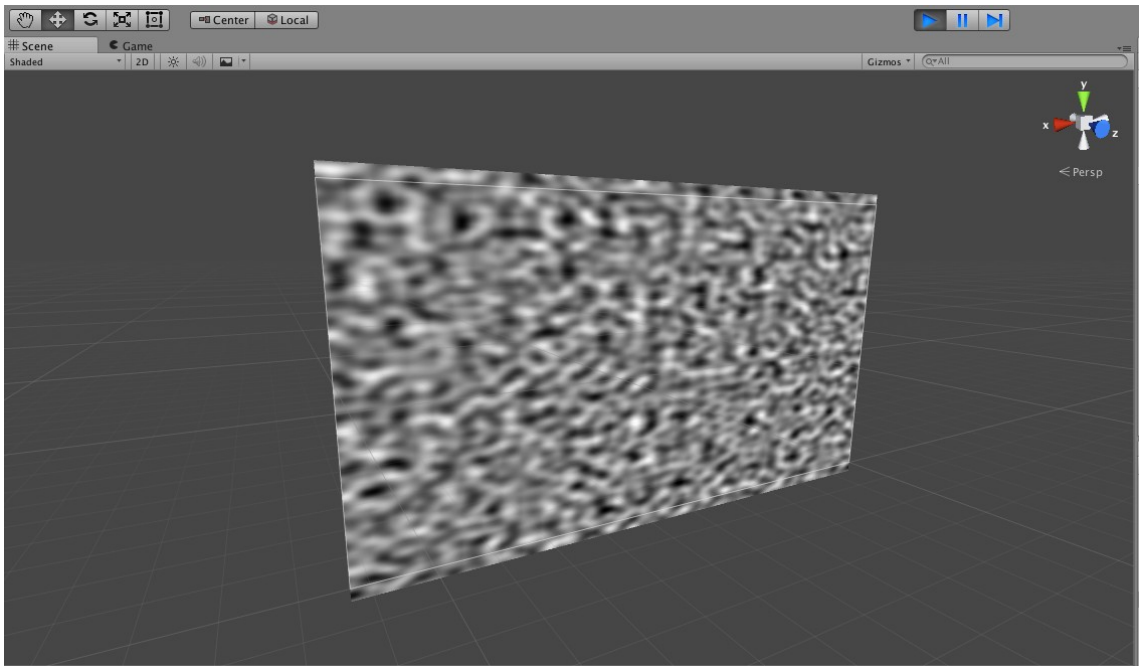
4.4.1 Perlin noise

Perlin noise -algoritmiin toteutettiin ensin Surflet-luokka. Surflet on 2 x 2 -ruudukkoalueen määritelmä, niin, että surfletin sisällä arvot seuraavat smoothstep-funktion ja väriliukuvektorien yhdistettyä arvoa, mutta ulkopuolella se on 0 (Kensler, 2015). Näin ollen jokaiselle 2 x 2 -ruudukkoalueelle voidaan määritellä oma surflet -objektinsa, ja siirtää laskemisvastuu vain ja ainoastaan niille surfleteille joilta tarvitaan tietoa. Surflet-objektit helpottavat myös reunanyli toiston laskemista, sillä yksi surflet voidaan määritellä niin, että se menee ruudukon rajan yli ja tulee toiselta puolelta lisäksi.

Laskemisvastuun siirto mahdollisti myös toteutuksen metodiin GetPerlin(Vector2 position), jonka Vector2-parametrille annetaan arvot x, y välillä 0-1. Ilman suurempia laskentaresursseja voidaan laskea vain tämän kohdan arvot. Tämä taas mahdollistaa sen, että algoritmin tuottamaa kohinaa voidaan tarpeen vaatiessa saada juuri niin monta arvoa kuin on tarve, ei

vähempää eikä enempää, ja se voidaan tehdä satunnaisesti. Näin ollen linkitys ja yhdistely muihin algoritmeihin helpottuu.

Toteutukseen lisättiin myös sivuajona tekstuurin täyttö. Tekstuurin täyttö käyttää suoraan Unityn omia Texture2D-luokan pikselinpiirtofunktioita. Reaaliaikainen näkymä yhdistettynä pikselinpiirtofunktioihin johti tekstuurin luontimetodin suureen hitauteen. Tämä ei kuitenkaan haittaa itse päätavoitetta, sillä algoritmi itsessään on tarpeeksi nopea, ja hidaskin tekstuuritäyttö toimii hyvänä visuaalisena varmistuksena algoritmin toteutuksen toimivuudesta (kuva 12).



Kuva 12. Perlin noise -algoritmin toteutuksen lopputulos Unity3D-pelimoottorin tekstuurissa.

4.4.2 Midpoint displacement

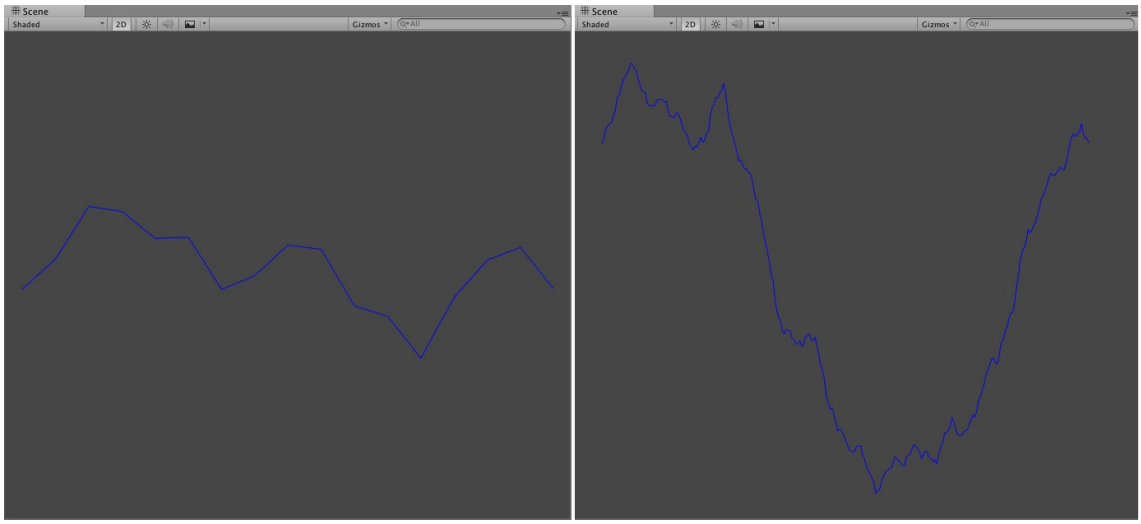
Midpoint displacement -algoritmin toteutus tehtiin laskemalla keskipistejako annetuista aloitus- ja lopetuspisteistä (Vector2). Lisäksi parametreinä luokan laskentametodille CalculateMidpointDisplacement() annetaan syvyys, eli kuinka monta kertaa keskipistejako tehdään, ja kovuusaste, joka tarkoittaa sitä, kuinka isoja erot voivat olla. Poiketen klassisesta midpoint displacement -algoritmista

määräytyy toteutetun algoritmin erojen maksimi aloitus- ja lopetuspisteen etäisyydestä, joka kerrotaan kovuusasteella.

Toteutus olisi voinut olla rekursiivinen, mutta se päätettiin tehdä listapohjaisesti. Unityssä on aiemmin törmätty rekursiivisissa funktioissa pinomuistin loppumiseen, joten ongelmien välttämiseksi tähän toteutukseen rekursiivista toteutusta ei tehty.

Algoritmi toteutettiin toimimaan niin, että käydään annetun syvyyden verran kierroksia läpi. Jokaisella syvyyskierroksella käydään kaikki tällä hetkellä mukana olevat pisteet läpi niin, että laitetaan uusien pisteiden listaan ensin mukana jo oleva piste, sitten lasketaan seuraavan pisteen ja tämän hetkisen pisteen välille midpoint displacement -algoritmilla piste, joka lisätään uusien pisteiden listaan myös.

Tämä toistetaan kunnes on jäljellä enää viimeinen piste, joka myös lisätään listaan. Tämän jälkeen käydään nämä uudet pisteet samalla tavalla läpi. Lopuksi metodi palauttaa kaikki lasketut pisteet. Nämä pisteet ovat algoritmin toteutuksen lopputulos (kuva 13).



Kuva 13. Midpoint displacement -algoritmin toteutuksen palauttama pistejoukko viivoina. Vasemmalla syvyydellä 4, kovuusasteella 0,5, oikealla syvyydellä 8, kovuusasteella 1.

Visuaaliseen debuggaukseen käytettiin tällä kerralla puhtaasti Unity Editorin työkaluja. Toisin kuin Perlin noise -algoritmin toteutuksessa, tämä visuaalinen representaatio ei näkyisi ruudulla mikäli kääntäisi ohjelman. Debuggaukseen käytin OnDrawGizmos()-metodia, jota Unity kutsuu editoriruudun päivitystä tehdessä mikäli komponentti-skriptiin on se määritelty. Unityn Gizmos -luokassa on myös määritelty helposti käytettäviä piirtofunktiota, jotka ovat omiaan visuaalisen debuggaukseen.

4.4.3 Space colonization

Space colonization -algoritmia varten kuvailtiin kaksi apuluokkaa, TreeNode ja AttractionPoint. TreeNode-luokka kuvastaa järjestelmässä olevaa pistettä, ja on saanut nimensä sekä itse alkuperäisen algoritmin tarkoituksesta – puiden proseduraalisen generoinnin – että lopputuloksen datan muodosta.

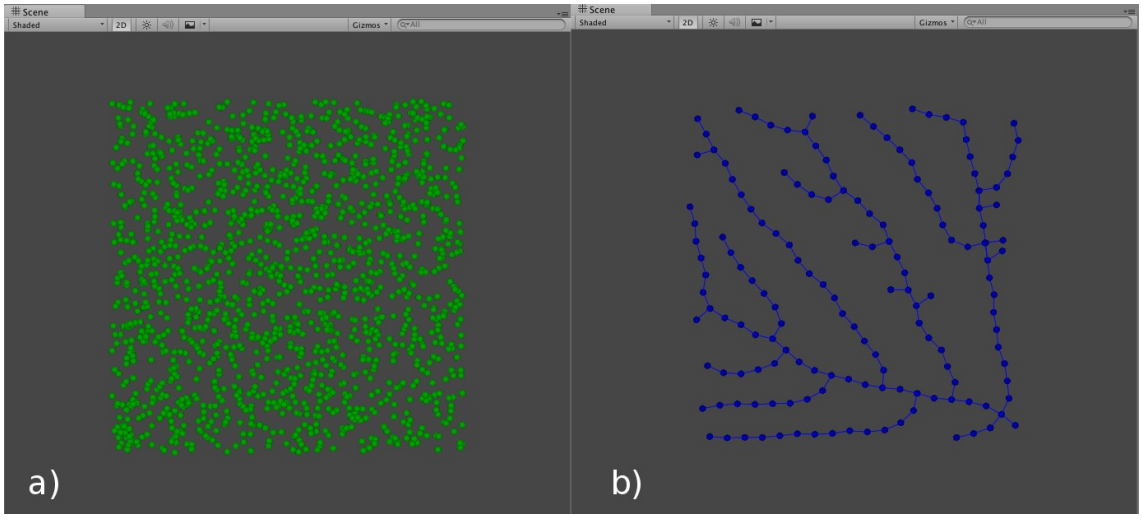
Apuluokkien sisälle rakennettiin yksinkertaista logiikkaa, joilla voidaan yksinkertaisesti hakea ja suodattaa sekä AttractionPoint-luokasta että TreeNode-luokasta vain niitä pisteitä, joihin voi tällä hetkellä vaikuttaa. Luokkien oli alun perin tarkoitus olla täysin atomisia ja lähes DTO-maisia, mutta itse generointifunktion selkiyttämiseksi lisättiin luokkaan luokkarajat ylittävää logiikkaa. Ohjelmakoodin pirstaloitumisen välttämiseksi, saavat nämä apuluokat pääluokan (SpaceColonization) instanssin dependency injection -ohjelmointitekniikalla olioita luodessa.

Toteutuksessa aluksi luodaan pistepilvi, joka koostuu siis AttractionPoint -objekteista. Pistepilven peittoalue on kiinteästi neliön muotoinen, joskin sen sijainti ja koko ovat vapaasti määriteltävissä (kuva 14a). Pistepilvi luodaan satunnaisesti, ja saatujen koordinaattien perusteella kysytään vapaasti määriteltävältä todennäköisyys-funktiolta, voiko tähän kohtaan luoda pistettä. Tämä funktio on vaihdettavissa SpaceColonization -luokkaa luodessa. Tämä

antaa mahdollisuuden linkittää esimerkiksi Perlin noise -algoritmin toteutus space colonization -algoritmin toteutukseen.

AttractionPoint-luokkaan toteutettiin ominaisuus, jonka hakemalla saadaan lähin vaikutusalueen sisäpuolella oleva TreeNode-objekti. Nämä TreeNode-objektit kerätään kaikista AttractionPoint-objekteista omaan listaansa. Tämä lista käydään läpi niin, että jokaisesta listan TreeNode-objektista haetaan jokaisen siihen vaikuttavan AttractionPoint-objektin normalisoidut suuntavektorit, joista otetaan keskiarvo. Tämä määrittelee seuraavaksi luotavan TreeNode-objektin paikan.

Kun kaikki vaikutettavat TreeNode-objektit on käyty tällä kierroksella lävitse, poistetaan tuhoamisalueella olevat AttractionPoint-objektit (eli ne, jotka ovat liian lähellä TreeNode -objekteja) ja haetaan kaikki AttractionPoint-objektien vaikutusalueella vaikutettavat TreeNodet. Mikäli on TreeNode-objekteja olemassa, joihin vaikutetaan, aloitetaan kierros uusiksi. Kun yhtään vaikutettavaa TreeNode -objektia tai yhtään AttractionPoint-objektia ei enää ole, on algoritmin työ valmis (kuva 14b).



Kuva 14. Space colonization -algoritmin toteutus. a) AttractionPoint-objektien sijainnit, eli peittoalueen pistepilvi. b) Lopullinen TreeNode-objekteista koostuva puurakenne

4.4.4 Dungeon room placement

Dungeon room placementiin eli luolastohuoneiden sijoitukseen käytettiin Nystromin (2014) kuvaamaa tapaa, jossa huoneiden määrän sijasta keskitytään yrityskertoihin. Toteutuksessa otetaan yrityskertojen maksimimäärän lisäksi parametreiksi myös maksimihuonemäärä, alue jonka sisälle huoneet luodaan sekä huoneen minimi- ja maksimikoko.

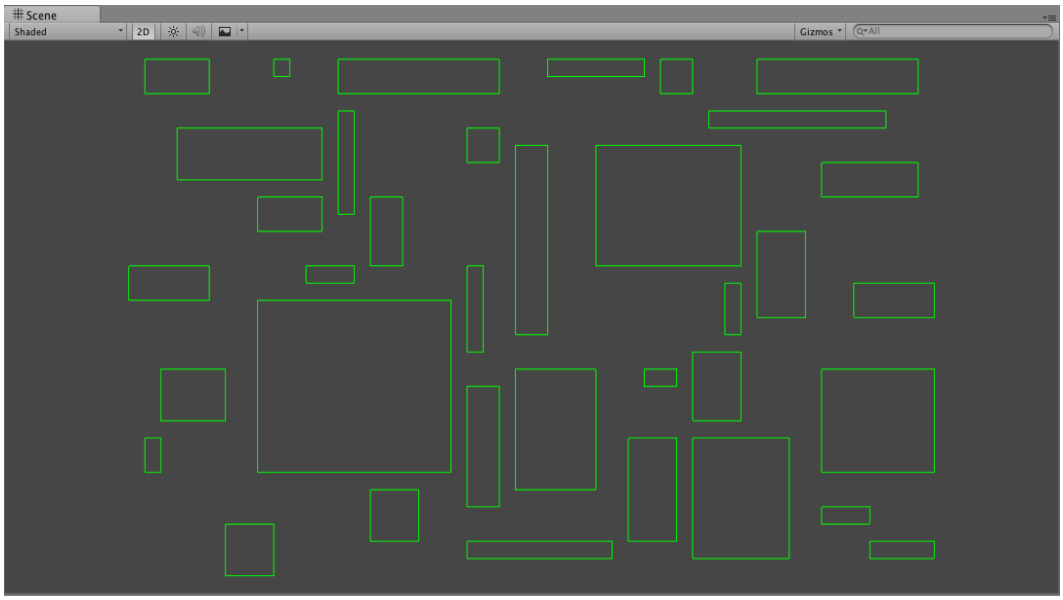
Mikäli huonemäärää ei ole määritelty tai se on 0, toimii järjestelmä aivan kuten Nystromin kuvaus algoritmista. Tällöin yhtä huonetta yritetään aina luoda tietty määrä kertoja, ja mikäli sopivaa huonetta ei saada luotua maksimiyrityskertojen sisällä, luovutetaan. Mikäli kuitenkin maksimihuonemäärä määritellään, ei toteutus luo yhtään enempää huoneita kuin mitä on määritelty. Myös maksimiyritysmäärä voi olla 0, eli määrittelemätön, jolloin huoneita luodaan niin kauan kuin maksimihuonemäärä täyttyy.

Huoneille voidaan määritellä ns. "askelkoko", joka asettaa rajat mahdollisille ko'oille. Mikäli askelkoko on 1, huoneet voivat sijaita ainoastaan kokonaislukujen mukaisesti, sekä huoneiden leveys ja korkeus voi olla vain

kokonaislukuja. Askelkoolla voidaan tarkemmin määritellä toteutus esimerkiksi muihin algoritmeihin sopivaksi.

Huoneiden etäisyys muista huoneista voidaan myös määritellä, joka sekin noudattaa askelkokoa. Näin ollen askelkoon ollessa yksi ja etäisyyden ollessa 1 on jokaisen huoneen seinän välillä vähintään yksi yksikkö. Tämäkin auttanee toteutuksen säätelyä vastaamaan yhdistämistä muiden algoritmien tarpeisiin.

Toteutuksen lopputulos on visualisoitu kuten space colonization -algoritmi ja midpoint displacement -algoritmi. Visuaalisessa debuggauksessa käytetään myös Unityn Gizmoja Editor-ikkunassa. Näin saadaan visuaalinen vahvistus toteutusvaiheessa algoritmin toiminnasta (kuva 15).



Kuva 15. Dungeon Room Generation -toteutus 200 maksimiyrityskerralla.

4.4.5 Maze generation

Toteutettavaksi maze generation -algoritmiksi valitun growing tree -algoritmin toteutus aloitettiin luomalla apuluokat MazeCell ja MazeWall. Koska muut toteutetut algoritmit eivät suoranaisesti toimi ruudukolla toisin kuin maze generation -algoritmit, tarvitsin tavan käsitellä lineaarista liukulukuavaruutta ikään kuin se olisi ruudukoitu. Toteutuksessa päädyttiin luomaan apuluokat

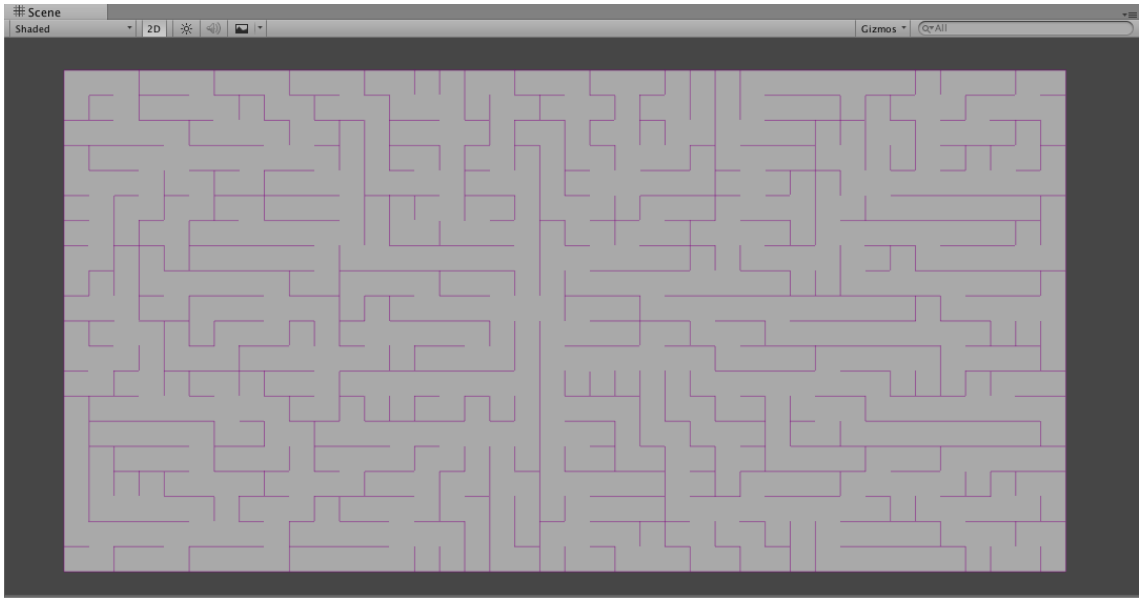
samoista syistä kuin SpaceColonizationissa käytetyt apuluokat, ja hyvin samankaltaisesti.

Molemmilla apuluokilla on pääsy pääluokkaan GrowingTree dependency injection -ohjelmointitekniikan avulla. Näissä kummassakin on logiikkaa, jonka avulla voidaan suodattaa ja lajitella sekä MazeCell-objektilistaa että MazeWall-objektilistaa.

Toteutuksessa luodaan ensin ruudukon MazeCell-objektit. Nämä objektit itsessään tietävät sijaintinsa, eikä se ole riippuvainen listasijainnista. Tämän jälkeen luodaan MazeWall-objekti joka kuvaa seinää sekä jokaisen vierekkäisen MazeCell -objektin välillä että ruudukon reunoissa. Tämän jälkeen lisätään yksi satunnainen MazeCell-objekti avoimien MazeCell-objektien listaan.

Aina yhdellä kierroksella valitaan avoimien MazeCell-objektien listasta valintafunktiolla yksi. Mikäli sillä ei ole yhtään vierailematonta naapuria, poistetaan se listasta ja aloitetaan uusi kierros. Jos sillä on vierailemattomia naapureita, valitaan niiden joukosta toisella valintafunktiolla, mikä näistä viereisistä, vierailemittomista soluista valitaan.

Kun seuraavana tuleva MazeCell-objekti on valittu, lisätään kyseinen objekti avointen listaan, merkataan se vierailluksi ja poistetaan tämänhetkisen ja seuraavan MazeCell-objektin välillä oleva MazeWall-objekti. Kun avointen lista on tyhjä, on sokkelo valmis (kuva 16).



Kuva 16. Growing tree -algoritmin toteutus käyttäen valintafunktiona satunnaisfunktiota

Molempia valintafunktioita muuttamalla voidaan määritellä valittujen MazeCell-objektien kriteerit täysin itsenäisesti. Kuten space colonization -algoritmin toteutuksessa, myös tässä toteutuksessa olisi voitu määritellä valintafunktiot parametrein. Tällainen toteutus olisi kuitenkin vienyt päätavoitteen toteutukselta aikaa, ja valintafunktioden muutos lopullista algoritmia varten pystyttiin toteuttamaan myös muilla tavoin. Tarvittavat valintafunktiot voidaan valita algoritmien yhdistelyvaiheessa, eikä niiden toteuttaminen riippumattomasti vaihdettaviksi tuota juuri nyt tarpeeksi lisäarvoa että se olisi kannattavaa.

4.5 Algoritmien yhdistäminen kartan luontiin

Algoritmien yhdistäminen jaettiin kahteen vaiheeseen. Ensimmäisessä vaiheessa yhdistettiin dungeon room placement- algoritmi maze generation -algoritmiin. Tästä muodostuu niin sanottu klassinen, perinteinen luolastongenerointimenetelmä.

Toisessa vaiheessa yhdistettiin perlin noise -algoritmi sekä space colonization -algoritmi keskenään. Tämän jälkeen ne yhdistettiin ensimmäisessä vaiheessa yhdisteltyyn klassiseen luolastogenerointimenetelmään.

4.5.1 Klassinen luolastogenerointi

Klassinen luolastogenerointi toteutettiin ensin luoden growing tree -algoritmin toteutuksessa käytetyt MazeCell- ja MazeWall-objektit (kuva 17a). Objektien luomisen jälkeen käytettiin dungeon room placement -algoritmia huoneet luomiseksi.

Näin saadut huoneet käytiin lävitse, ja kaikki MazeCell-objektit merkattiin growing tree -algoritmin ohjaamiseksi jo käydyiksi soluiksi. Sen jälkeen kaikki MazeWall-objektit poistettiin mikäli niiden molemmilla puolilla olevat MazeCell-objektit olivat merkattu jo käydyiksi. Näin saatiin poistettua huoneiden sisällä olevat MazeWall-objektit sekä poistettua huoneiden sisällä olevat solut maze generation -algoritmin luomisesta (kuva 17b).

Huoneen luomisen ja sokkelosolujen merkitsemisen jälkeen ajettiin huoneiden reunasolujen tunnistus. Jokaiselle solulle merkittiin kaikki vierekkäiset huoneet. Huoneiden reunasolujen tunnistuksen jälkeen jokaisen huoneen reunasolujen joukosta valittiin satunnaisesti yhdestä neljään solua, jonka yhteydessä oleva seinä jo käydyksi merkattuun (eli tässä vaiheessa ainoastaan jokin huoneen sisällä olevista sokkelosoluista), kyseisessä huoneessa olevaan soluun poistettiin. Näin saatiin yhdistetyä tuleva sokkelo sekä jokainen huone. Tämä varmistaa sen, että jokainen huone on saavutettavissa.

Yhtenä vaihtoehtona huoneiden poistamiseksi maze generation -algoritmin laskemisesta pohdittiin sekä kokeiltiin asianomaisten MazeCell- ja MazeWall-objektien poistamista. Tämä kuitenkin johti siihen, että huoneen reunasolujen tunnistus ja huoneen sisään vievän seinän poisto meni lähes mahdottomaksi.

Lisäksi pitäytyen koko alueen kattavassa, growing tree -algoritmin toteutuksen datarakenteissa, saatiiin helpommin toteutettua luotujen karttarakenteiden tuominen testiympäristöön pelattaviksi komponenteiksi.

Huoneiden huomioonottamisen jälkeen itse growing tree -algoritmi ajettiin. Aloitussolun valintaa täytyi hiukan muokata niin, että valikoidaan satunnaisesti mikä tähänsä jo käymätön solu. Sen jälkeen algoritmin annettiin ajaa kuten se oli implementoitukin.

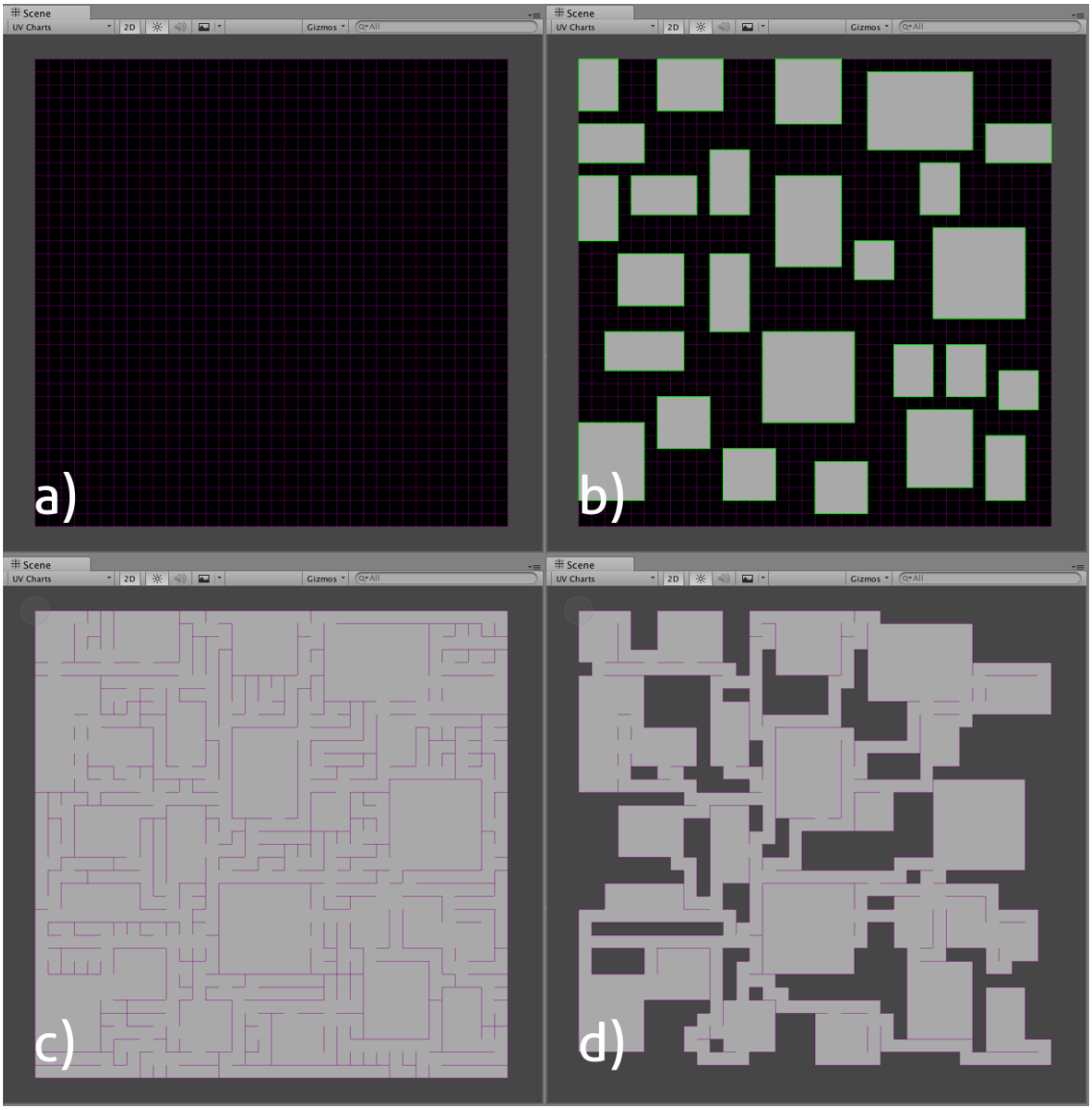
Growing tree -algoritmin ajamisen jälkeen luolasto oli lähes valmis. Se oli kuitenkin hyvin ahdas; jokainen solu on liitetty sokkeloon (kuva 17c). Aluetta avattiin hiukan toteuttamalla growing tree -algoritmiin lisäksi vielä Nystromin (2014) kuvaaman dead end removal -metodin kaltainen toiminto.

Toisin kuin Nystromin implementaatiossa, ei tässä toteutuksessa ole mitään rajaa kuinka paljon umpikujia metodi poistaa. Tämä toteutus käy kaikki umpikujat yksi kerrallaan, ja mikäli umpikuja löytyy - eli MazeCell-objekti jolla on tasan 3 seinää – lisätään puuttuva seinä, ja poistetaan solu kokonaan. Poistossa solu tuhoamisen sijasta poistetaan olemassaolevien solujen listasta ja lisätään poistettujen listaan, visualisoinnin helpottamiseksi niin debuggaustarkoituksissa kuin myös testiympäristössä.

Mikäli sokkelosta ei olisi merkattu pois huoneita, eikä niitä olisi liitetty sokkeloon, tämä umpikujanpoisto johtaisi itseasiassa siihen, että vain ja ainoastaan yksi sokkelosolu jäisi. Kyseisen sokkelosolun kohta riippuisi täysin siitä, missä järjestyksessä soluja käsitellään. Koska tässä implementaatiossa on kuitenkin mukana myös huoneita jotka on liitetty sokkeloon, johtaa tämä siihen, että ainoastaan huoneiden väliset käytävät jäävät sokkeloon.

Yhdistelyn lopputuloksena saadaan aikaisiksi huoneita sisältävä luolastogeometria. Huoneet ovat yhdistetty sokkelonomaisilla käytävillä, ja

käytävien sekä huoneiden välillä on myös tilaa, eikä se ole (subjektiivisesti ajateltuna) liian ahdas (kuva 17d).

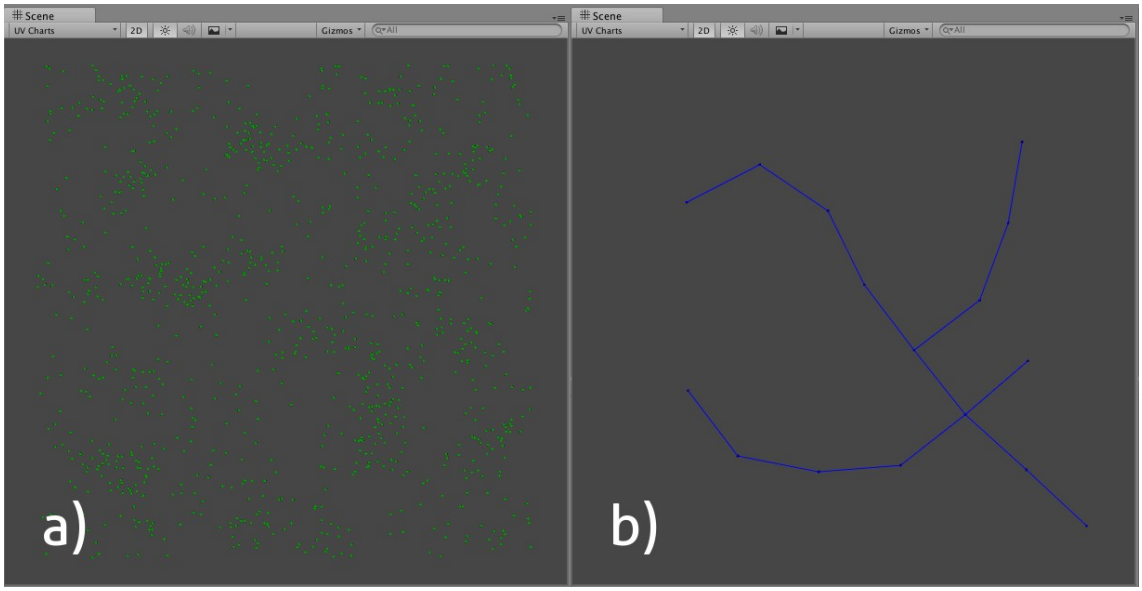


Kuva 17. Klassinen luolastogeneroinnin vaiheita. a) Ruudukon luonti MazeCell- ja MazeWall-objektein b) Huoneiden luonti dungeon room placement -algoritmilla ja niiden merkintä käydyiksi c) Huoneet ovat yhdistetty sokkeloon ja sokkelo luotu growing tree -algoritmilla. d) Umpikujien poisto dead end removal -metodilla.

4.5.2 Yhdistelty luolastogenerointi

Seuraavaksi yhidtettiin muut työssä implementoidut sisällöntuotantoalgoritmit klassiseen luolastogenerointialgoritmiin. Midpoint displacement -algoritmia ei mielikuvituksen puutteessa saatu luontevasti yhdistymään muihin algoritmeihin ollenkaan. Alkuperäinen ajatus huoneiden seinien muuntamisesta luolastomaisemmaksi vaatisi testiympäristön vahvaa muokkausta. Tämä siis jätettiin yhdistelystä kokonaan pois. Yhdistelyyn tuli kuitenkin mukaan perlin noise -algoritmi sekä space colonization -algoritmi.

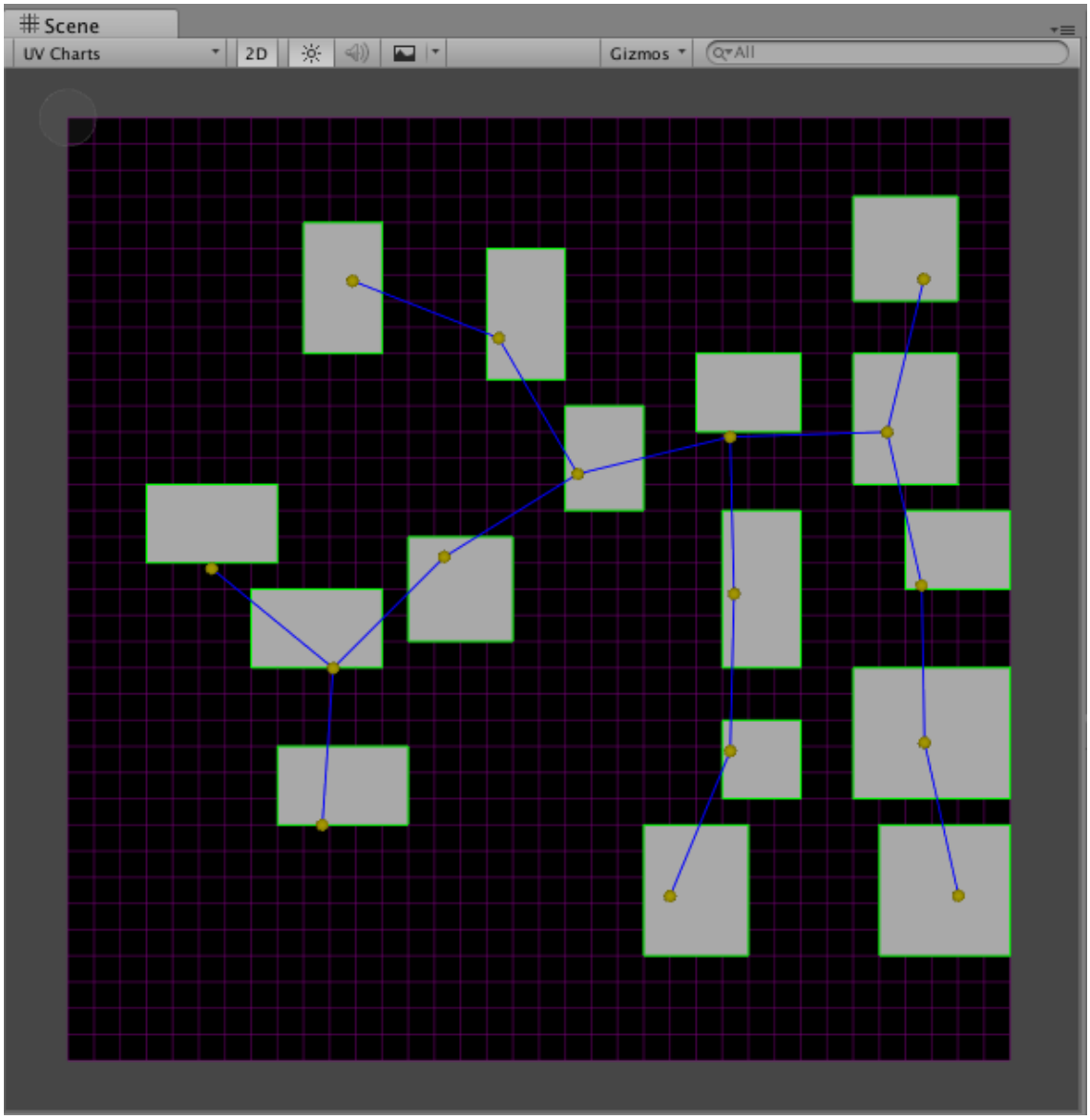
Ensin yhdistettiin perlin noise -algoritmi space colonization -algoritmin AttractionPoint-objektien luomisen todennäköisyysfunktioksi. Tämän jälkeen ajettiin space colonization -algoritmi toteutuksensa mukaisesti (kuva 18a, b).



Kuva 18. Space colonization -algoritmi. a) Houkutuspisteinä toimivat AttractionPoint-objektit luotu käyttäen Perlin noise -algoritmia todennäköisyysfunktiota b) Lopullinen algoritmin luoma puurakenne

Näin saadut järjestelmäpisteet miellettiin huoneiden sijainniksi. Dungeon room placement -algoritmia täytyi muokata luomaan huoneet pisteiden lähettyville. Algoritmi mukautettiin niin, että lähes kaikissa tapauksissa luotu huone sisältää

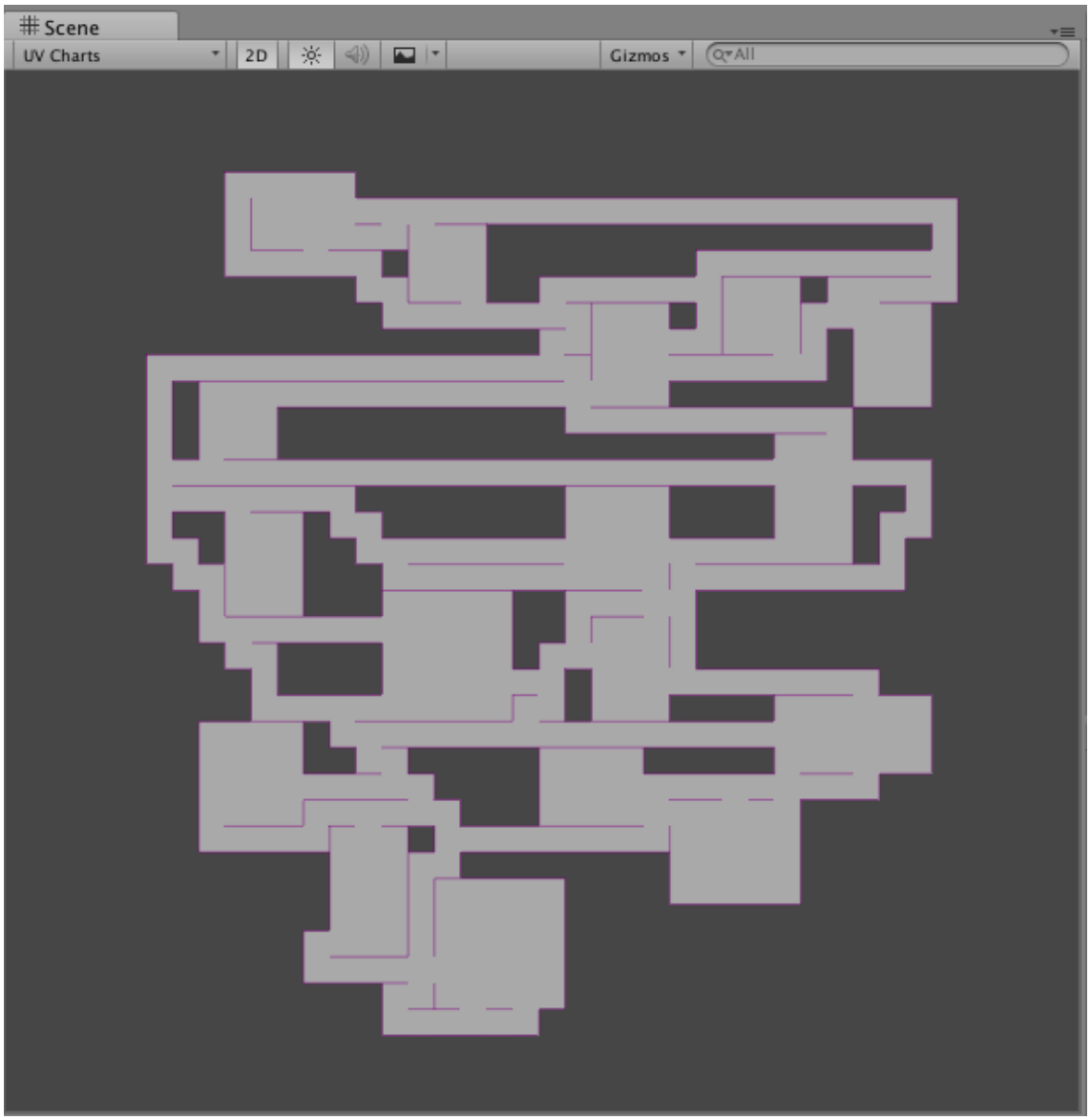
pyydetyn pisteen, ja mikäli ei sisällä, se on vain yhden askellusarvon verran sivussa (kuva 19).



Kuva 19. Huoneiden lisääminen space colonization -algoritmin luomien järjestelmäpisteiden mukaisesti

Huoneiden luomisen jälkeen jatkettiin kuten klassisessa luolaston generoinnissa (kuva 20). Space colonization -algoritmilla luotu puurakenne antaisi oivan mahdollisuuden toteuttaa käytävien yhdistely niin, että ainoastaan yhdistettyihin järjestelmäpisteisiin luotujen huoneiden välille toteutettasiin sokkelonluonti. Tämän kuitenkin todettiin tarpeettomaksi – klassisen luolastogeneroinnin

sokkelonluontialgoritmi on riittävä yhdistämään huoneet toisiinsa, kuitenkin antaen mahdollisuuden space colonization -algoritmille tuoda rakennetta huoneiden sijaintien kautta.



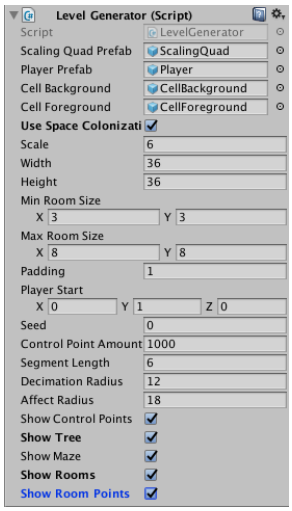
Kuva 20. Lopullinen yhdisteltyjen ja muokattujen algoritmien tuottaman kenttägeometria

Maze generation -algoritmin solunvalinta muutettiin tässä vaiheessa vastaamaan enemmän metroidvania-kenttien geometriaa. Metroidvania-kentissä on tyypillisesti enemmän vaakasuoraista käytävää kuin pystysuoraista, joten solunvalinta valitsee aggressiivisesti aina solut niin, että maksimoidaan

sivuille meneminen. Tässä tavassa tuli esille myös sivuvaikutuksena metroidvania-kentille tyypillinen pystykäytävän rakenne, jossa molemmilla puolilla vuorotellen on tasot jonne hyppiä (kuva 21).

Kuva 21. Sivuvaikutuksena luotu Metroidvania-kentille tyypillinen pystykäytävärakenne.

Space colonization -algoritmin kohdentaminen, sekä huoneiden minimi- ja maksimikoon sekä muiden dungeon room placement -algoritmiin että growing tree -algoritmiin vaikuttavien parametrien muutosmahdollisuus toteutettiin niin, että niitä pystyi muokkaamaan Unity Editorin Inspector-ikkunassa. Tämä mahdollisti oikeiden arvojen helpomman etsimisen, jotta toivottu lopputulos voitasiin saada tai ainakin päästä lähelle sitä (kuva 22).



Kuva 22. Unity Editorin Inspector-ikkunassa voidaan muokata algoritmien asetuksia.

4.6 Testiympäristö

Testiympäristössä testattiin kuinka lopulta kehitetyt kenttägeometrian luontiin tarkoitetut implementaatiot voitaisiin siirtää abstraktista datamuodosta pelattavaan muotoon pelimoottorin sisälle. Testiympäristöön toteutettiin kenttägeometrian lisäksi pelaajahahmo, jota voidaan liikuttaa pelikentän sisällä kuten tyypillisissä metroidvania-peleissä, lukuun ottamatta muita pelillisiä ominaisuuksia. Erikoiskykynä pelaajahahmo voi tehdä hyppyjä myös ilmassa, ja hyppiä voi loputtomasti. Näin kenttiä voidaan tutkia, vaikka mukana ei olekaan erillisiä hyppimiseen tarkoitettavia tasoja, vaan pelkästään pelkkä tavallinen kenttägeometria.

Graafisesti testiympäristö on erittäin yksinkertainen. Valikon ulkoasua on paranneltu käytettävyyden kohentamisen nimissä, vaikkakin tähän tutkimukseen ei ole tarpeen edes käyttää käännettyä versioita. Koska testiympäristön oli kuitenkin tarkoitus olla pelattava, tehtiin myös käännetty versio. Jatkossa testiympäristöä voisi mahdollisesti kehittää eteenpäin, niin graafisesti kuin pelillisesti.

4.6.1 Kenttägeometria

Ensimmäisessä versiossa kaikki huoneet, joissa pystyttäisiin olemaan (mukaanlukien käytävät) mielettiin neliömäisiksi polygoneiksi. Oletettiin, että mikäli nämä kaikki polygonit yhdistetään toisiinsa, pystytään luomaan pelkästään jokaisen polygonin verteksin välillä oleva seinä. Nämä seinät toimisivat sitten pelimoottorin sisällä liikkumista estävinä, eli toimisivat kenttägeometriana kentän sisällä.

Seinän luontia varten toteutettiin ScalingQuad -luokka, joka toimii yhdessä Unityn GameObject-objektien kanssa. Unity-maailmassa se on siis normaali

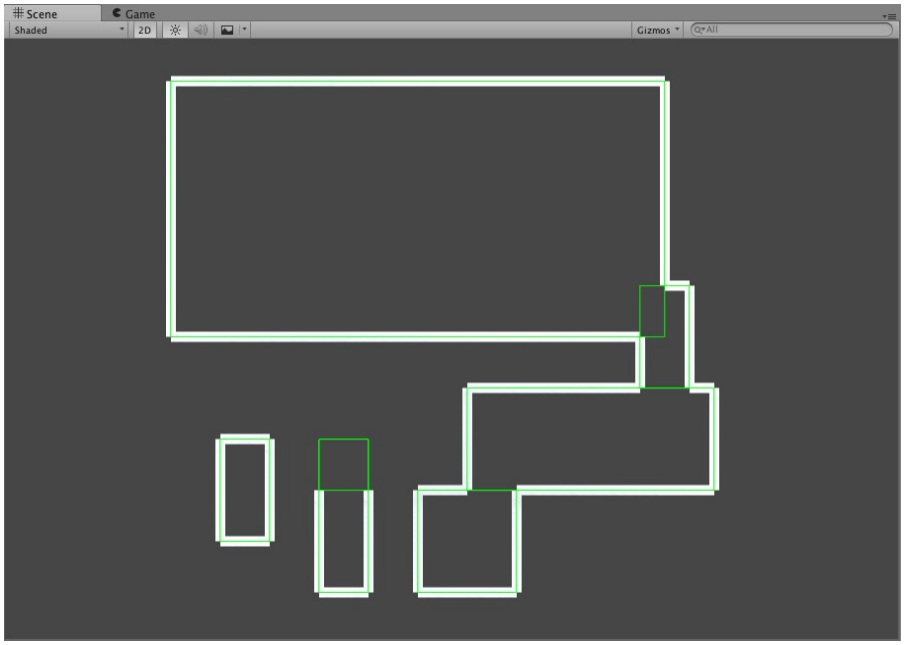
MonoBehaviour-luokasta perivä aliluokka, ja sen saa liitetty komponentiksi GameObject-objektiin.

Luokan periaate on yksinkertainen: luokalle annetaan kaksi pistettä. Objekti sijoitetaan pisteiden keskivälille, minkä jälkeen pisteiden väliin muodostuvan janan kulma suhteessa x-akseliin lasketaan. Sen jälkeen käännetään objektia z-akselilla kulman verran, ja asetetaan x-akselin skaalaksi pisteiden etäisyys.

Tällä tavoin graafinen komponentti, johon ScalingQuad-luokka lisätään, voidaan erikseen määritellä, eikä sitä tarvitse luoda tyhjästä. ScalingQuad-luokasta luotiin prefab eli GameObject-objekti jonka asetukset tallennetaan tiedostoon. Jokaiselle pisteparille luotiin uusi objekti tästä prefabista.

Seinätoteutus toimi, mutta huoneiden yhdistelyä ei edellä kuvaillulla tavalla saatu toimimaan. Kun huoneet mielletään polygoneiksi, niiden yhdistäminen on loogisinta tehdä polygonien yhdistämiseen tarkoitetulla boolean-operaatiolla. Tässä tapauksessa kyseeseen tulisi AND-operaation.

AND-operaatiota lähdettiin toteuttamaan geenerisessä mielessä niin, että se toimisi myös muun muotoisille polygoneille kuin neliölile, pohjustaen midpoint displacement -algoritmin käyttöä. Kehityksessä törmättiin kuitenkin ongelmaan: mikäli yhdistettävien polygonien verteksien pisteet ovat toistensa edgeillä, tai mikä vielä pahempaa, yhdistettävissä polygoneissa on molemmilla verteksi samassa sijainnissa, tulee vastaan lukematon määrä erikoistilanteita, jotka täytyisi ottaa huomioon (kuva 23).



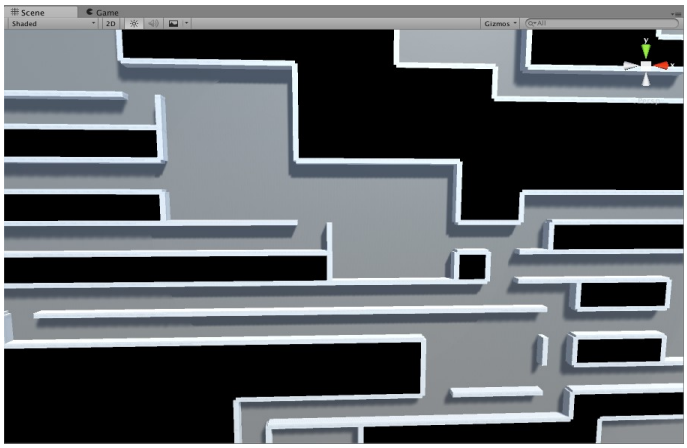
Kuva 23. Polygon clipping -algoritmin toteutuksen yrityksen lopputulos. Vihreällä huoneet, valkoisella yhdistetty polygoni.

Asiaa tutkittiin ja huomattiin, että AND-operaation toteutus oli tehty hyvin samankaltaisesti kuin Greiner-Hormannin polygon clipping -algoritmi. Greiner-Hormannin polygon clipping -algoritmissa päällekkäisyydet on ratkaistu mitä mielenkiintoisimmalla tavalla: pisteet, jotka ovat päällekkäin tai edgeillä, siirretään. Siirto on hyvin pieni, joten visuaalisesti sitä ei pitäisi nähdä, mutta tarpeeksi iso, että algoritmi toimii. (Greiner & Hormann, 1998)

Lisäksi ongelmaksi muodostui reikien käsittely. Toteutuksessa ei oltu ollenkaan otettu huomioon, mitä tehdä kun yhdisteltävät polygonit kiertävät niin, että keskellä jäisi tyhjää. Tapa tuoda kenttägeometria pelattavaan muotoon on tämän tutkimuksen päätavoitteen kannalta toissijainen, joten jatkossa käytettiin maze generation -algoritmin solu- ja seinärakenteita yhdistettynä jo luotuun ScalingQuad-prefabiin tuomaan kenttägeometria testiympäristöön.

Maze generation -algoritmin solu- ja seinärakenteita voidaan käyttää lähes suoraan kenttägeometriaa pelattavaan muotoon luodessa. Jokaista seinää kohden voidaan kopioida ScalingQuad-prefab, ja koska seinässä on valmiiksi sijaintidata seinän alulle ja lopulle, nämä pisteet saatiin asettetua suoraan.

Lisäksi soluja varten luotiin uusi prefab, joka tehtiin toimimaan ikään kuin huoneen taustaseinänä. Myös poistetuille solulle luotiin prefab-objekti, ja näin ollen saatiin myös etummaisena olevan seinän kohdille grafiikka auttamaan erottamaan solut, jotka eivät kuulu kenttägeometrian sisälle (kuva 24).



Kuva 24. Kenttägeometria tuotu runkopeliin MazeWall- ja MazeCell-objektien avulla

4.6.2 Pelaajahahmo

Pelaajahahmon tarkoituksena on toimia testiympäristössä tapana, jolla voidaan tutkia kenttien toimivuutta ja vertailla sitä tyypilliseen metroidvania-kenttiin, kriteerien mukaisesti. Toteutuksessa kiinnitettiin erityistä huomiota liikkeen sulavuuteen.

Pelaajahahmo luotiin käyttäen Unityn omaa CharacterController -luokkaa. Näin pelaajahahmon logiikka pysyi yksinkertaisena ja helppona toteuttaa. Kuten muunkin testiympäristön, pelaajahahmon graafinen ulkoasu on hyvin riisuttu (kuva 25). Vaikka se ei ole visuaalisesti paras ratkaisu, se kuitenkin mahdollisti kenttägeometrian helpomman tutkimisen. Kenttien tutkimisen helpottamiseksi ilman tasoja, pelaajahahmoon toteutettiin mahdollisuus hyppiä ilmassa.

Kuva 25. Testiympäristön pelaajahahmo kaikessa loistossaan.

5 Tulokset

Tulokset yllättivät positiivisesti. Henkilökohtainen tavoite onnistua toteuttamaan erilaisia sisällöntuotantoalgoritmeja ja perinteisiä luolastongenerointimenetelmiä hyväksikäyttäen kenttägeometria 2D-tasohyppelypelille onnistuttiin saavuttamaan.

Testiympäristön alkuperäinen tavoite oli kehittää se toimivaksi peliksi. Valitettavasti sitä ei kuitenkaan voitu toteuttaa ajan puutteen vuoksi. Se ei kuitenkaan ollut työn päätarkoitus, ja alkuperäisen tavoitteen mukaisesti testiympäristön tärkein osa tuli toteutettua: todeta generoitujen kenttien toimivuus kontrolloitavan pelaajahahmon kautta.

5.1 Klassinen vai yhdistelty luolastogenerointi

Eräs työn tavoitteista oli verrata klassista, jo peleissä käytössä olevaa luolastongenerointimenetelmää ja yhdisteltyä algoritmia keskenään. Tarkoituksena oli myös verrata molemmilla tavoilla toteutettuja kenttiä metroidvania-kentästä analysoituja kriteerejä vastaan.

Toteutetussa klassisen luolastogenerointimetodin tuottamissa kentissä oli paljon huoneita ja vähän käytävätilaa. Käytävät olivat usein erittäin sokkelomaisia.

Asetuksia muuttamalla saatiin huonemäärä pienemmäksi ja huoneiden väli väljemmäksi, mutta tämä teki kentästä silti hyvin tasaisesti täytetyn.

Tiiviyden todettiin metroidvania-kenttiä analysoitessa olevan lajityypillinen kenttägeometrian ominaisuus, joten sen suhteen klassinen luolastongenerointi voisi toimia ihan hyvin. Käytävien sokkelomaisuus ei välttämättä ole kuitenkaan lajityypille sopiva.

Yhdisteltyjen algoritmien tuottamat kentät toisaalta taas olivat paljon rakenteellisempia. Ne eivät täyttäneet koko pelialaa tasaisesti, kuten klassinen tapa. Tämä rakenteellisuus toi kuitenkin kenttään enemmän persoonallisuutta, ja oli sen myös senkin tähden metroidvania-kenttämäinen.

Lisäksi yhdistelmäalgoritmeihin toteutettu muutos growing tree -algoritmin solunvalintametodissa, jolla aggressiivisesti valittiin aina samalla x-akselilla oleva solu mikäli vain mahdollista, johti pitkiin vaakasuoriin käytäviin, joka on paljon metroidvania-kenttiin sopivampi vaihtoehto kuin erittäin sokkeloiset käytävät.

Valintametodin muutoksen sivuvaikutuksena tuli myös tyypillisesti metroidvania-kentissä esiintyvä pystysuoran huoneen käyttötapa. Usein metroidvania-kenttien pystysuorissa huoneissa on molemmin puolin vuorotellen tasot, joilla pelaaja pääsee liikkumaan.

Tämä muutos kuitenkin toteuttiin growing tree -algoritmissa, joten sen pystyy ottamaan käyttöön myös klassiseen luolastongenerointiin. Algoritmien yhdistelmää käyttämällä kuitenkin saadaan lisäksi huoneiden sijoittelurakenne, huolimatta siitä, että toteutuksesta puuttuu space colonization -algoritmin luoman puurakenteen mukaiset huoneliitokset.

Perlin noise -algoritmin käyttö space colonization -algoritmin pistepilven todennäköisyysfunktion tuoma lisäarvo jää arvoitukseksi. Teoriassa space

colonization -algoritmin luoman puurakenteen pitäisi ohjautua täysin pistepilven mukaisesti, ja niin se tekeekin. Siitä huolimatta puurakenteen erot käytettynä perussatunnaisfunktiota suhteessa perlin noise -algoritmin tuottamaan satunnaisuuteen ovat hyvin pienet. Perlin noise -algoritmin tuoma arvo kentän generointiin tai space colonization -algoritmin tuottaman puurakenteen ohjaukseen on siis epäselvä. Tämä vaatisi lisätutkimuksia jotka menevät tämän tutkimuksen alkuperäisten tavoitteiden ulkopuolelle.

Midpoint displacement -algoritmi ei lopulta tullut millään tavalla mukaan algoritmien yhdistämiseen. Sen ollessa yksinkertainen algoritmi, sen pitäisi kyllä jollain tavalla olla hyödyllinen, mutta lisähyöty ei pakosti ole kentän generoinnissa. Tässä työssä siitä ei ainakaan ollut lopulta mitään lisäarvoa.

Kaikki tulokset huomioon ottaen tutkimuksen perusteella voidaan väittää, että varsinkin space colonization -algoritmin käyttö antaa lisäarvoa 2D-tasohyppelipelikenttien generointiin verraten pelkästään klassisten luolastogenerointimenetelmien käyttöön. Muokkaamalla maze generation -algoritmia ja käyttämällä spae colonization -algoritmia kentän generoinnissa saadaan hyvin Metroidvania-kenttien kaltaisia rakenteita, joskaan ei aivan täysin samanlaisia.

6 Pohdinta

Projektia voisi halutessaan kehittää eteenpäin. Space colonization -algoritmi soveltui hyvin antamaan huoneiden sijainnille rakennetta. Perlin noise -algoritmin tuoma hyöty on kyseenalainen, joten mikäli sitä haluaisi käyttää, pitäisi sen hyödyllisyys arvioida tarkemmin ja perusteellisemmalla tutkimuksella.

Growing tree -algoritmi oli hyvä valinta maze generation -algoritmeista, sen monipuolisudeen tähden. Muokkaus aggressiiviseen vaakasuuntaisuuteen

solunvalinnassa osoittautui hyväksi ratkaisuksi. Jatkossa solunvalintaa voisi kehittää esimerkiksi niin, että suositaan mahdollisimman suoria ja pitkiä käytäviä, ei vain pelkästään vaakasuuntaisuutta. Valintafunktion voisi myös muuttaa olemaan käymättä soluissa, joilla on useampi kuin 1 naapuri, ja näin saataisiin väljyyttä käytävien sijainteihin. Tämä kuitenkin poistaisi sivuvaukutuksena tulevat hyödylliset rakenteet.

Kentänluonnin toteuttavan algoritmien yhdistelmän ja pelimaailmaan tuonnin pystyisi paketoimaan ja esimerkiksi tarjoamaan Unity Asset Storessa. Tämä vaatisi paketoinnin suhteen hiukan lisää työtä, joka ei suoranaisesti liity kehitykseen.

Testiympäristön pelattavaksi peliksi kehittämisen suhteen olisi tarpeellista ottaa kenttägeometrian luontiin myös tasot, joita ei tässä työssä otettu huomioon. Erilliset tasot huoneiden ja käytävien sisällä mahdollistaisi perinteisemmän pelaajahahmon liikkumisen. Lisäksi testiympäristön pelillistämiseksi pitäisi totetuttaa vähintään alku ja loppu, ja mahdollisesti myös viholliset sekä tapa taistella vihollisten kanssa.

6.1 Haasteet

Testiympärsitön työstössä suurimmaksi tekniseksi ongelmaksi ilmeni polygon clipping -algoritmin toteutus. Haaste kasvoi itseasiassa niin suureksi, että kesken kehitystyön täytyi toteutus jättää kesken, ja etsiä jokin toinen tapa toteuttaa kenttägeometrian abstraktin datan muunto pelissä toimivaksi. Mikäli näin ei olisi tehty, olisi jo valmiiksi rajoitettu ja koko ajan lisää rajoittuva aikataulu estänyt käytännön työn valmistumisen.

Alkuperäisestä odotuksesta poiketen, työstö VSCode -editorilla, Unity3D:n kokeellisella Linux-versiolla ja pelkästään Linuxilla sekä raportin kirjoittaminen että käytännön osion tekeminen ei itseasiassa edes ollut haaste, vaikka niin

alun perin ajateltiin. Sitä mahdollisuutta varten, että työskentely ei onnistu Linux-käyttöjärjestelmällä, kehiteltiin vara-vaihtoehtoja, mutta niiden käyttöä ei lopulta edes tarvinnut miettiä.

Haasteeksi osoittautui myös tiukassa aikataulussa ohjelmoinnin toteutus. Koska aikaa oli rajallisesti, kaikki toteutusta edistävä toiminta koostui koodista, jonka tarkoituksena oli puhdas tavoittellisuus, eikä elegantti, helposti muokattavissa oleva ohjelmakoodi. Tämä johti moniin tilanteisiin, joissa toteutuksen logiikka petti, ja asioita jouduttiin kirjoittamaan uusiksi, jotta ne toimisivat vähän loogisemmin. Joissakin tapauksissa tämä johti sietämättömän huonoon suorituskykyyn.

Esimerkiksi MazeWall-solukkoa luotaessa growing tree -algoritmin ensimmäisessä toteutuksessa tarkistettiin jokaiselta solun reunalta onko siellä seinä jo olemassa vai ei, ja sen jälkeen luotiin vasta seinä. Tämä johti siihen, että jo esimerkiksi 40 x 40 -alue kesti useita kertoja kauemmin luoda kuin esimerkiksi 20 x 20 -alue. Vaikka suorituskyky ei ollut tämän tutkimuksen pääaihe, täytyy suorituskyvyn kuitenkin olla sellainen, että testaaminen on mahdollista. Tämä suorituskykyongelma johti turhaan kehitysajan hukkaan ongelmaa metsästäessä, vaikka ongelma lopullisessa toteutuksessa onkin korjattu.

6.2 Onnistumiset

Kenttägeometrian generointi sekä perinteisillä luolastongenerointimenetelmillä että yhdistelemellä erilaisia muita sisällöntuotantoalgoritmeja onnistui erittäin hyvin. Toteutusvaiheessa oli muutamia vaikeuksia siellä täällä, jotka kuitenkin ylitettiin tai kierrettiin. Algoritmien toteutus Unity3D-moottorille onnistui erinomaisen hyvin.

Kenttägeometrian muunto abstraktista datasta pelattavaan muotoon onnistui vähän heikommin. Alkuperäinen tarkoitus oli rakentaa boolean AND-operaatiolla iso yhdistetty polygoni, jonka reunat kuvastaisivat seiniä, jotka luotaisiin. Tämä tapa aiheutti kuitenkin liikaa aikapaineita toteutuksessa, ja koska tapa, jolla kenttägeometriadata tuodaan peliin ei ollut merkitsevä, se jouduttiin poistamaan. Tilalle kuitenkin onneksi keksittiin käyttää maze generation -algoritmiin luotuja datarakenteita, joiden ansiosta kentän luonti pelimaailmaan onnistui.

6.3 Epäonnistumiset

Suurin epäonnistuminen oli virhearviointi midpoint displacement -algoritmin mahdollisuudesta yhdistää mukaan kenttägeometrian luontiin. MazeWall-objektin kuvaamia seiniä olisi mahdollisesti voitu muokata algoritmillä, mutta ongelmat runkopeliin vientiin olisivat taas palautuneet polygon clipping -algoritmin toteuttamiseen. Muutos olisi myös ollut lähinnä kosmeettinen, eikä näin ollen olisi vaikuttanut kenttägeometrian rakenteeseen itsessään.

Metroidvania-kenttien analysointi ja niistä kriteerien määrittely olisi voinut olla myös tarkempaa. Nykyisessä analysoinnissa keskityttiin vain huoneisiin ja käytäviin, mutta tarkemmalla analyysillä olisi voitu esimerkiksi huomata, että mukana on myös kaltevia tasoja.

6.4 Yhteenveto

Yleiset sisällöntuotannon generointimenetelmät eivät suoranaisesti sovellu 2D-tasohyppelipelikenttien geometrian luontiin. Kuitenkin pienin muutoksin joko algoritmeihin tai lopputuloksiin, tai yhdistelemällä algoritmeja toisiinsa, on näillä geneerisimmilläkin algoritmeilla mahdollisuus toteuttaa mielekästä sivustakuvattua 2D-kenttää.

Varsinkin space colonization -algoritmin käyttöönotto johti huomattavaasti rakenteellisempaan lopputulokseen. Tätä algoritmia voisi helposti käyttää myös jatkokehityksessä. Perlin noise -algoritmin hyödyt jäivät vajavaisiksi, joten niitä ei tämän työn perusteella ainakaan kannata juuri tähän tarkoitukseen käyttää, ainakaan ilman lisätutkimuksia.

Midpoint displacement -algoritmi jäi täysin käyttämättä. Sitä mahdollisesti voisi käyttää lopullisen kenttägeometrian rajojen kohinan lisäämisessä. Oikeilla parametreillä tällä tavoin voisi esimerkiksi mallintaa luolastojen seinämiä. Jos algoritmia muokattaisiin niin, että muutosakseli olisi poikittainen suhteessa muutettavaan janaan ja muutosmäärä määräytyisi janan pituuden sekä muutoskertoimen mukaan, voisi tarpeen vaatiessa minkä tahansa janan jakaa kerran ja toistettavasti. Näin voitaisiin esimerkiksi jakaa esimerkiksi satunnainen määrä janoja, eikä niiden tarvitsisi olla selkeällä linjalla. Valitettavasti tämä toteutus jäi teknisten ongelmien ja aikarajoitteiden takia toteuttamatta ja kokeilematta, mutta se voisi jatkokehityksessa olla ihan aiheellinen testata.

Pelkästään maze generation -algoritmiin tehdyt soluvalinnan muutokset vaikuttivat suunnattomasti kentän rakenteellisuuden vaikutelmaan. Lopputulokset ovat subjektiivisia, joskin hiukan enemmän Metroidvania-kentille tyypillisempiä kuin satunnainen soluvalinta.

Työssä ja tutkimuksessa huomattiin, että algoritmeja yhdistellen ja muokaten voidaan generoida 2D-tasohyppelypelikenttiä, jotka eivät nojaudu vain templatingiin tai muuten johda kovin toistuviin kenttärakenteisiin. Mahdollisuudet mielekkäisiin kenttiin ovat monipuoliset ja todella mielenkiintoiset, mutta koska tässä työssä keskityttiin vain kenttägeometriaan, vaatisivat tämän työn aikana toteutetut metodit vielä jatkokehitystä toimiakseen itsenäisinä kenttinä.

Lähteet

Buck, J. 2011a. Maze Generation: Aldous-Broder Algorithm. http://weblog.jamisbuck.org/2011/1/17/maze-generation-aldous-broder-algorithm. 23.2.2016.

Buck, J. 2011b. Maze Generation: Algorithm Recap. http://weblog.jamisbuck.org/2011/2/7/maze-generation-algorithm-recap. 20.2.2016.

Buck, J. 2011c. Maze Generation: Prim's Algorithm. http://weblog.jamisbuck.org/2011/1/10/maze-generation-prim-s-algorithm. 20.2.2016.

Buck, J. 2011d. Maze Generation: Growing Tree Algorithm. http://weblog.jamisbuck.org/2011/1/27/maze-generation-growing-tree-algorithm. 20.2.2016.

Fournier, A., Fussell, D. & Carpenter, L. 1982. Computer rendering of stochastic models. Communications of the ACM. Volume 25, Issue 6, June 1982.

Greiner, G., Hormann, K. 1998. Efficient clipping of arbitrary polygons. https://www.researchgate.net/publication/220184531_Efficient_Clipping_of_Arbitrary_Polygons. 27.4.2016.

Johnson, J. 2016. A study of generative systems for modeling natural phenomena. http://valhallawebdesign.com/Thesis/. 6.2.2016.

Kensler, A. 2015. Building Up Perlin Noise. http://eastfarthing.com/blog/2015-04-21-noise/. 6.2.2016.

Nystrom, R. 2014. Rooms and Mazes: A Procedural Dungeon Generator. http://journal.stuffwithstuff.com/2014/12/21/rooms-and-mazes/. 20.2.2016.

Perlin, K. 1999. Making noise. http://www.noisemachine.com/talk1/index.html. 6.2.2016.

Runions, A., Fuhrer, M., Lane, B., Federl, P., Rolland-Lagan, A.-G., Prusinkiewicz, P. 2005. Modeling and visualization of leaf venation patterns. http://algorithmicbotany.org/papers/colonization.egwnp2007.html. 6.2.2016.

Runions, A., Lane, B. & Prusinkiewicz, P. 2007. Modeling Trees with a Space Colonization Algorithm. http://algorithmicbotany.org/papers/colonization.egwnp2007.large.pdf. 6.2.2016.

Tsai, M.-F. G. 2003. Fractal Landscapes. http://www.sfu.ca/~rpyke/335/projects/tsai/report1.htm. 6.2.2016.